conference
proceedings

# 19th USENIX Security Symposium

***Washington, DC***
***August 11–13, 2010***

Sponsored by
The USENIX Association
usenix

USENIX
Proceedings of the 19th USENIX Security Symposium
Washington, DC August 11–13, 2010

ISBN 978-1-931971-77-5

**USENIX Association**

# Proceedings of the
# 19th USENIX Security Symposium

**August 11–13, 2010**
**Washington, DC**

# Conference Organizers

## Program Chair

Ian Goldberg, *University of Waterloo*

## Program Committee

Lucas Ballard, *Google, Inc.*
Adam Barth, *University of California, Berkeley*
Steven M. Bellovin, *Columbia University*
Nikita Borisov, *University of Illinois at Urbana-Champaign*
Bill Cheswick, *AT&T Labs—Research*
George Danezis, *Microsoft Research*
Rachna Dhamija, *Harvard University*
Vinod Ganapathy, *Rutgers University*
Tal Garfinkel, *VMware and Stanford University*
Jonathon Giffin, *Georgia Institute of Technology*
Steve Gribble, *University of Washington*
Alex Halderman, *University of Michigan*
Cynthia Irvine, *Naval Postgraduate School*
Somesh Jha, *University of Wisconsin*
Samuel King, *University of Illinois at Urbana-Champaign*
Negar Kiyavash, *University of Illinois at Urbana-Champaign*
David Lie, *University of Toronto*
Michael Locasto, *George Mason University*
Mohammad Mannan, *University of Toronto*
Niels Provos, *Google, Inc.*
Reiner Sailer, *IBM T.J. Watson Research Center*
R. Sekar, *Stony Brook University*
Hovav Shacham, *University of California, San Diego*
Micah Sherr, *University of Pennsylvania*
Patrick Traynor, *Georgia Institute of Technology*
David Wagner, *University of California, Berkeley*
Helen Wang, *Microsoft Research*
Tara Whalen, *Office of the Privacy Commissioner of Canada*

## Invited Talks Committee

Dan Boneh, *Stanford University*
Sandy Clark, *University of Pennsylvania*
Dan Geer, *In-Q-Tel*

## Poster Session Chair

Patrick Traynor, *Georgia Institute of Technology*

## The USENIX Association Staff

# External Reviewers

Mansour Alsaleh
Elli Androulaki
Sruthi Bandhakavi
David Barrera
Sandeep Bhatkar
Mihai Christodorescu
Arel Cordero
Weidong Cui
Drew Davidson
Lorenzo De Carli
Brendan Dolan-Gavitt
Matt Federikson
Adrienne Felt
Murph Finnicum
Simson Garfinkel
Phillipa Gill
Xun Gong
Bill Harris
Matthew Hicks
Peter Honeyman
Amir Houmansadr
Joshua Juen
Christian Kreibich
Louis Kruger
Marc Liberatore
Lionel Litty
Jacob Lorch
Daniel Luchaup
David Maltz
Joshua Mason
Kazuhiro Minami
Prateek Mittal
David Molnar
Fabian Monrose
Tyler Moore
Alexander Moshchuk
Shishir Nagaraja
Giang Nguyen
Moheeb Abu Rajab
Wil Robertson
Nabil Schear
Jonathan Shapiro
Kapil Singh
Abhinav Srivastava
Shuo Tang
Julie Thorpe
Wietse Venema
Qiyan Wang
Scott Wolchok
Wei Xu
Fang Yu
Hang Zhao

# 19th USENIX Security Symposium
# August 11–13, 2010
# San Jose, CA, USA



## Wednesday, August 11

### Protection Mechanisms



### Privacy



### Detection of Network Attacks

## Thursday, August 12

### Dissecting Bugs



### Cryptography



### Internet Security



### Real-World Security





## Friday, August 13

### Web Security



### Securing Systems



### Using Humans

# Message from the Program Chair

I would like to start by thanking the USENIX Security community for making this year's call for papers the most successful yet. We had 207 papers originally submitted; this number was reduced to 202 after one paper was withdrawn by the authors, three were withdrawn for double submission, and one was withdrawn for plagiarism. This was the largest number of papers ever submitted to USENIX Security, and the program committee faced a formidable task.

Each PC member reviewed between 20 and 22 papers (with the exception of David Wagner, who reviewed an astounding 27 papers!) in multiple rounds over the course of about six weeks; many papers received four or five reviews.

We held a two-day meeting at the University of Toronto on April 8–9 to discuss the top 76 papers. This PC meeting ran exceptionally smoothly, and I give my utmost thanks to the members of the PC; they were the ones responsible for such a pleasant and productive meeting. I would especially like to thank David Lie and Mohammad Mannan for handling the logistics of the meeting and keeping us all happily fed.

By the end of the meeting, we had selected 30 papers to appear in the program—another record high for USENIX Security. The quality of the papers was outstanding. In fact, we left 8 papers on the table that we would have been willing to accept had there been more room in the program. This year's acceptance rate (14.9%) is in line with the past few years.

The USENIX Security Symposium schedule offers more than refereed papers. Dan Boneh, Sandy Clark, and Dan Geer headed up the invited talks committee, and they have done an excellent job assembling a slate of interesting talks. Patrick Traynor is the chair of this year's poster session, and Carrie Gates is chairing our new rump session. This year we decided to switch from the old WiP (work-in-progress) session to an evening rump session, with shorter, less formal, and (hopefully) some funnier entries. Carrie has bravely agreed to preside over this experiment. Thanks to Dan, Sandy, Dan, Patrick, and Carrie for their important contributions to what promises to be an extremely interesting and fun USENIX Security program.

Of course this event could not have happened without the hard work of the USENIX organization. My great thanks go especially to Ellie Young, Devon Shaw, Jane-Ellen Long, Anne Dickison, Casey Henderson, Tony Del Porto, and board liaison Matt Blaze. Because USENIX takes on the task of running the conference and attending to the details, the Program Chair and the Program Committee can concentrate on selecting the refereed papers.

Finally, I would like to thank Fabian Monrose, Dan Wallach, and Dan Boneh for convincing me to take on the role of USENIX Security Program Chair. It has been a long process, but I am very pleased with the results.

Welcome to Washington, D.C., and the 19th USENIX Security Symposium. I hope you enjoy the event.

**Ian Goldberg, University of Waterloo**
**Program Chair**

# Adapting Software Fault Isolation to Contemporary CPU Architectures

David Sehr Robert Muth Cliff Biffle Victor Khimenko Egor Pasko
Karl Schimpf Bennet Yee Brad Chen

{*sehr,robertm,cbiffle,khim,pasko,kschimpf,bsy,bradchen*}*@google.com*

## Abstract

Software Fault Isolation (SFI) is an effective approach to sandboxing binary code of questionable provenance, an interesting use case for native plugins in a Web browser. We present software fault isolation schemes for ARM and x86-64 that provide control-flow and memory integrity with average performance overhead of under 5% on ARM and 7% on x86-64. We believe these are the best known SFI implementations for these architectures, with significantly lower overhead than previous systems for similar architectures. Our experience suggests that these SFI implementations benefit from instruction-level parallelism, and have particularly small impact for workloads that are data memory-bound, both properties that tend to reduce the impact of our SFI systems for future CPU implementations.

## 1 Introduction

As an application platform, the modern web browser has some noteworthy strengths in such areas as portability and access to Internet resources. It also has a number of significant handicaps. One such handicap is computational performance. Previous work [30] demonstrated how software fault isolation (SFI) can be used in a system to address this gap for Intel 80386-compatible systems, with a modest performance penalty and without compromising the safety users expect from Web-based applications. A major limitation of that work was its specificity to the x86, and in particular its reliance on x86 segmented memory for constraining memory references. This paper describes and evaluates analogous designs for two more recent instruction set implementations, ARM and 64-bit x86, with pure software-fault isolation (SFI) assuming the role of segmented memory.

The main contributions of this paper are as follows:

- A design for ARM SFI that provides control flow and store sandboxing with less than 5% average overhead,
- A design for x86-64 SFI that provides control flow and store sandboxing with less than 7% average overhead, and
- A quantitative analysis of these two approaches on modern CPU implementations.

We will demonstrate that the overhead of fault isolation using these techniques is very low, helping to make SFI a viable approach for isolating performance critical, untrusted code in a web application.

### 1.1 Background

This work extends Google Native Client [30].[1] Our original system provides efficient sandboxing of x86-32 browser plugins through a combination of SFI and memory segmentation. We assume an execution model where untrusted (hence sandboxed) code is multi-threaded, and where a trusted runtime supporting OS portability and security features shares a process with the untrusted plugin module.

The original NaCl x86-32 system relies on a set of rules for code generation that we briefly summarize here:

- The code section is read-only and statically linked.
- The code section is conceptually divided into fixed sized bundles of 32 bytes.
- All *valid* instructions are reachable by a disassembly starting at a bundle beginning.
- All indirect control flow instructions are replaced by a multiple-instruction sequence (*pseudo-instruction*) that ensures target address alignment to a bundle boundary.
- No instructions or pseudo-instructions in the binary crosses a bundle boundary.

All rules are checked by a verifier before a program is executed. This verifier together with the runtime system comprise NaCls trusted code base (TCB).

For complete details on the x86-32 system please refer to our earlier paper [30]. That work reported an average overhead of about 5% for control flow sandboxing, with the bulk of the overhead being due to alignment considerations. The system benefits from segmented memory to avoid additional sandboxing overhead.

Initially we were skeptical about SFI as a replacement for hardware memory segments. This was based in part on running code from previous research [19], indicating about 25% overhead for x86-32 control+store SFI, which we considered excessive. As we continued

[1] We abbreviate Native Client as "NaCl" when used as an adjective.

our exploration of ARM SFI and sought to understand ARM behavior relative to x86 behavior, we could not adequately explain the observed performance gap between ARM SFI at under 10% overhead with the overhead on x86-32 in terms of instruction set differences. With further study we understood that the prior implementations for x86-32 may have suffered from suboptimal instruction selection and overly pessimistic alignment.

Reliable disassembly of x86 machine code figured largely into the motivation of our previous sandbox design [30]. While the challenges for x86-64 are substantially similar, it may be less clear why analogous rules and validation are required for ARM, given the relative simplicity of the ARM instruction encoding scheme, so we review a few relevant considerations here. Modern ARM implementations commonly support 16-bit Thumb instruction encodings in addition to 32-bit ARM instructions, introducing the possibility of overlapping instructions. Also, ARM binaries commonly include a number of features that must be considered or eliminated by our sandbox implementation. For example, ARM binaries commonly include read-only data embedded in the text segment. Such data in executable memory regions must be isolated to ensure it cannot be used to invoke system call instructions or other instructions incompatible with our sandboxing scheme.

Our architecture further requires the coexistence of trusted and untrusted code and data in the same process, for efficient interaction with the trusted runtime that provides communications and portable interaction with the native operating system and the web browser. As such, indirect control flow and memory references must be constrained to within the untrusted memory region, achieved through sandboxing instructions.

We briefly considered using page protection as an alternative to memory segments [26]. In such an approach, page-table protection would be used to prevent the untrusted code from manipulating trusted data; SFI is still required to enforce control-flow restrictions. Hence, page-table protection can only avoid the overhead of data SFI; the control-flow SFI overhead persists. Also, further use of page protection adds an additional OS-based protection mechanism into the system, in conflict with our requirement of portability across operating systems. This OS interaction is complicated by the requirement for multiple threads that transition independently between untrusted (sandboxed) and trusted (not sandboxed) execution. Due to the anticipated complexity and overhead of this OS interaction and the small potential performance benefit we opted against page-based protection without attempting an implementation.

## 2 System Architecture

The high-level strategy for our ARM and x86-64 sandboxes builds on the original Native Client sandbox for x86-32 [30], which we will call NaCl-ARM, NaCl-x86-64, and NaCl-x86-32 respectively. The three approaches are compared in Table 1. Both NaCl-ARM and NaCl-x86-64 sandboxes use alignment masks on control flow target addresses, similar to the prior NaCl-x86-32 system. Unlike the prior system, our new designs mask high-order address bits to limit control flow targets to a logical zero-based virtual address range. For data references, stores are sandboxed on both systems. Note that reads of secret data are generally not an issue as the address space barrier between the NaCl module and the browser protects browser resources such as cookies.

In the absence of segment protection, our ARM and x86-64 systems must sandbox store instructions to prevent modification of trusted data, such as code addresses on the trusted stack. Although the layout of the address space differs between the two systems, both use a combination of masking and guard pages to keep stores within the valid address range for untrusted data. To enable faster memory accesses through the stack pointer, both systems maintain the invariant that the stack pointer always holds a valid address, using guard pages at each end to catch escapes due to both overflow/underflow and displacement addressing.

Finally, to encourage source-code portability between the systems, both the ARM and the x86-64 systems use ILP32 (32-bit Int, Long, Pointer) primitive data types, as does the previous x86-32 system. While this limits the 64-bit system to a 4GB address space, it can also improve performance on x86-64 systems, as discussed in section 3.2.

At the level of instruction sequences and address space layout, the ARM and x86-64 data sandboxing solutions are very different. The ARM sandbox leverages instruction predication and some peculiar instructions that allow for compact sandboxing sequences. In our x86-64 system we leverage the very large address space to ensure that most x86 addressing modes are allowed.

## 3 Implementation

### 3.1 ARM

The ARM takes many characteristics from RISC microprocessor design. It is built around a load/store architecture, 32-bit instructions, 16 general purpose registers, and a tendency to avoid multi-cycle instructions. It deviates from the simplest RISC designs in several ways:

- condition codes that can be used to predicate most instructions
- "Thumb-mode" 16-bit instruction extensions can improve code density



| Feature | NaCl-x86-32 | NaCl-ARM | NaCl-x86-64 |
|---|---|---|---|
| Addressable memory | 1GB | 1GB | 4GB |
| Virtual base address | Any | 0 | 44GB |
| Data model | ILP32 | ILP32 | ILP32 |
| Reserved registers | 0 of 8 | 0 of 15 | 1 of 16 |
| Data address mask method | None | Explicit instruction | Implicit in result width |
| Control address mask method | Explicit instruction | Explicit instruction | Explicit instruction |
| Bundle size (bytes) | 32 | 16 | 32 |
| Data embedded in text segment | Forbidden | Permitted | Forbidden |
| "Safe" addressing registers | All | `sp` | `rsp`, `rbp` |
| Effect of out-of-sandbox store | Trap | No effect (typically) | Wraps mod 4GB |
| Effect of out-of-sandbox jump | Trap | Wraps mod 1GB | Wraps mod 4GB |

Table 1: Feature Comparison of Native Client SFI schemes. NB: the current release of the Native Client system have changed since the first report [30] was written, where the addressable memory size was 256MB. Other parameters are unchanged.

- relatively complex barrel shifter and addressing modes

While the predication and shift capabilities directly benefit our SFI implementation, we restrict programs to the 32-bit ARM instruction set, with no support for variable-length Thumb and Thumb-2 encodings. While Thumb encodings can incrementally reduce text size, most important on embedded and handheld devices, our work targets more powerful devices like notebooks, where memory footprint is less of an issue, and where the negative performance impact of Thumb encodings is a concern. We confirmed our choice to omit Thumb encodings with a number of major ARM processor vendors.

Our sandbox restricts untrusted stores and control flow to the lowest 1GB of the process virtual address space, reserving the upper 3GB for our trusted runtime and the operating system. As on x86-64, we do not prevent untrusted code from *reading* outside its sandbox. Isolating faults in ARM code thus requires:

- Ensuring that untrusted code cannot execute any forbidden instructions (e.g. undefined encodings, raw system calls).
- Ensuring that untrusted code cannot store to memory locations above 1GB.
- Ensuring that untrusted code cannot jump to memory locations above 1GB (e.g. into the service runtime implementation).

We achieve these goals by adapting to ARM the approach described by Wahbe et al. [28]. We make three significant changes, which we summarize here before reviewing the full design in the rest of this section. First, we reserve no registers for holding sandboxed addresses, instead requiring that they be computed or checked in a single instruction. Second, we ensure the integrity of multi-instruction sandboxing *pseudo-instructions* with a variation of the approach used by our earlier x86-32 system [30], adapted to further prevent execution of embedded data. Finally, we leverage the ARM's fully predicated instruction set to introduce an alternative data address sandboxing sequence. This alternative sequence replaces a data dependency with a control dependency, preventing pipeline stalls and providing better overhead on multiple-issue and out-of-order microarchitectures.

#### 3.1.1 Code Layout and Validation

On ARM, as on x86-32, untrusted program text is separated into fixed-length *bundles*, currently 16 bytes each, or four machine instructions. All indirect control flow must target the beginning of a bundle, enforced at runtime with address masks detailed below. Unlike on the x86-32, we do not need bundles to prevent overlapping instructions, which are impossible in ARM's 32-bit instruction encoding. They are necessary to prevent indirect control flow from targeting the interior of pseudo-instruction and bundle-aligned "trampoline" sequences. The bundle structure also allows us to support data embedded in the text segment, with data bundles starting with an invalid instruction (currently `bkpt 0x7777`) to prevent execution as code.

The validator uses a fall-through disassembly of the text to identify valid instructions, noting the interior of pseudo-instructions and data bundles are not valid control flow targets. When it encounters a direct branch, it further confirms that the branch target is a valid instruction. For indirect control flow, many ARM opcodes can cause a branch by writing `r15`, the program counter. We forbid most of these instructions[2] and consider only explicit branch-to-address-in-register forms such as `bx r0` and their conditional equivalents. This restriction is consistent with recent guidance from ARM for compiler

[2] We do permit the instruction `bic r15, rN, MASK` Although it allows a single-instruction sandboxed control transfer, it can have poor branch prediction performance.

writers. Any such branch must be immediately preceded by an instruction that masks the destination register. The mask must clear the most significant two bits, restricting branches to the low 1GB, and the four least significant bits, restricting targets to bundle boundaries. In 32-bit ARM, the Bit Clear (`bic`) instruction can clear up to eight bits rotated to any even bit position. For example, this pseudo-instruction implements a sandboxed branch through `r0` in eight bytes total, versus the four bytes required for an unsandboxed branch:

```
bic r0, r0, #0xc000000f
bx r0
```

As we do not trust the contents of memory, the common ARM return idiom `pop {pc}` cannot be used. Instead, the return address must be popped into a register and masked:

```
pop { lr }
bic lr, lr, #0xc000000f
bx lr
```

Branching through LR (the link register) is still recognized by the hardware as a return, so we benefit from hardware return stack prediction. Note that these sequences introduce a data dependency between the `bx` branch instruction and its adjacent masking instruction. This pattern (generating an address via the ALU and immediately jumping to it) is sufficiently common in ARM code that the modern ARM implementations [3] can dispatch the sequence without stalling.

For stores, we check that the address is confined to the low 1GB, with no alignment requirement. Rather than destructively masking the address, as we do for control flow, we use a `tst` instruction to *verify* that the most significant bit is clear together with a predicated store:[3]

```
tst r0, #0xc0000000
streq r1, [r0, #12]
```

Like `bic`, `tst` uses an eight-bit immediate rotated to any even position, so the encoding of the mask is efficient. Using `tst` rather than `bic` here avoids a data dependency between the guard instruction and the store, eliminating a two-cycle address-generation stall on Cortex-A8 that would otherwise triple the cost of the added instruction. This illustrates the usefulness of the ARM architecture's fully predicated instruction set. Some predicated SFI stores can also be synthesized in this manner, using sequences such as `tsteq`/`streq`. For cases where the compiler has selected a predicated store that cannot be synthesized with `tst`, we revert to a `bic`-based sandbox, with the consequent address-generation stall.

[3]The `eq` condition checks the Z flag, which `tst` will set if the selected bit is clear.

We allow only base-plus-displacement addressing with immediate displacement. Addressing modes that combine multiple registers to compute an effective address are forbidden for now. Within this limitation, we allow all types of stores, including the Store-Multiple instruction and DMA-style stores through coprocessors, provided the address is checked or masked. We allow the ARM architecture's full range of pre- and post-increment and decrement modes. Note that since we mask only the base address and ARM immediate displacements can be up to $\pm$4095 bytes, stores can access a small band of memory outside the 1GB data region. We use guard pages at each end of the data region to trap such accesses.[4]

### 3.1.2 Stores to the Stack

To allow optimized stores through the stack pointer, we require that the stack pointer register (SP) always contain a valid data address. To enforce this requirement, we initialize SP with a valid address before activating the untrusted program, with further requirements for the two kinds of instructions that modify SP. Instructions that update SP as a side-effect of a memory reference (for example `pop`) are guaranteed to generate a fault if the modified SP is invalid, because of our guard regions at either end of data space. Instructions that update SP directly are sandboxed with a subsequent masking instruction, as in:

```
mov SP, r1
bic SP, SP, #c0000000
```

This approach could someday be extended to other registers. For example, C-like languages might benefit from a frame pointer handled in much the same way as the SP, as we do for x86-64, while Java and C++ might additionally benefit from efficient stores through `this`. In these cases, we would also permit moves between any two such data-addressing registers without requiring masking.

### 3.1.3 Reference Compiler

We have modified LLVM 2.6 [13] to implement our ARM SFI design. We chose LLVM because it appeared to allow an easier implementation of our SFI design, and to explore its use in future cross-platform work. In practice we have also found it to produce faster ARM code than GCC, although the details are outside the scope of this paper. The SFI changes were restricted to the ARM target implementation within the `llc` binary, and required approximately 2100 lines of code and table modifications. For the results presented in this paper we used the compiler to generate standard Linux executables with access to the full instruction set. This allows us to isolate the behavior of our SFI design from that of our trusted runtime.

[4]The guard pages "below" the data region are actually at the top of the address space, where the OS resides, and are not accessible from user mode.



## 3.2 x86-64

While the mechanisms of our x86-64 implementation are mostly analogous to those of our ARM implementation, the details are very different. As with ARM, a valid data address range is surrounded by guard regions, and modifications to the stack pointer (`rsp`) and base pointer (`rbp`) are masked or guarded to ensure they always contain a valid address. Our ARM approach relies on being able to ensure that the lowest 1GB of address space does not contain trusted code or data. Unfortunately this is not possible to ensure on some 64-bit Windows versions, which rules out simply using an address mask as ARM does. Instead, our x86-64 system takes advantage of more sophisticated addressing modes and use a small set of "controlled" registers as the base for most effective address computations. The system uses the very large address space, with a 4GB range for valid addresses surrounded by large (multiples of 4GB) unmapped/protected regions. In this way many common x86 addressing modes can be used with little or no sandboxing.

Before we describe the details of our design, we provide some relevant background on AMD's 64-bit extensions to x86. Apart from the obvious 64-bit address space and register width, there are a number of performance relevant changes to the instruction set. The x86 has an established practice of using related names to identify overlapping registers of different lengths; for example `ax` refers to the lower 16-bits of the 32-bit `eax`. In x86-64, general purpose registers are extended to 64-bits, with an `r` replacing the `e` to identify the 64 vs. 32-bit registers, as in `rax`. x86-64 also introduces eight new general purpose registers, as a performance enhancement, named `r8 - r15`. To allow legacy instructions to use these additional registers, x86-64 defines a set of new prefix bytes to use for register selection. A relatively small number of legacy instructions were dropped from the x86-64 revision, but they tend to be rarely used in practice.

With these details in mind, the following code generation rules are specific to our x86-64 sandbox:

- The module address space is an aligned 4GB region, flanked above and below by protected/unmapped regions of 10$\times$4GB, to compensate for scaling (c.f. below)
- A designated register "RZP" (currently `r15`) is initialized to the 4GB-aligned base address of untrusted memory and is read-only from untrusted code.
- All `rip` update instructions must use `RZP`.

To ensure that `rsp` and `rbp` contain a valid data address we use a few additional constraints:

- `rbp` can be modified via a copy from `rsp` with no masking required.
- `rsp` can be modified via a copy from `rbp` with no masking required.
- Other modifications to `rsp` and `rbp` must be done with a pseudo-instruction that post-masks the address, ensuring that it contains a valid data address.

For example, a valid `rsp` update sequence looks like this:

```
%esp = %eax
lea (%RZP, %rsp, 1), %rsp
```

In this sequence the assignment[5] to `esp` guarantees that the top 32-bits of `rsp` are cleared, and the subsequent add sets those bits to the valid base. Of course such sequences must always be executed in their entirety. Given these rules, many common store instructions can be used with little or no sandboxing required. `Push, pop` and `near call` do not require checking because the updated value of `rsp` is checked by the subsequent memory reference. The safety of a store that uses `rsp` or `rbp` with a simple 32-bit displacement:

```
mov disp32(%rsp), %eax
```

follows from the validity invariant on `rsp` and the guard ranges that absorb the displacement, with no masking required. The most general addressing expression for an allowed store combines a valid base register (`rsp`, `rbp` or `RZP`) with a 32-bit displacement, a 32-bit index, and a scaling factor of 1, 2, 4, or 8. The effective address is computed as:

```
basereg + indexreg * scale + disp32
```

For example, in this pseudo-instruction:

```
add $0x00abcdef, %ecx
mov %eax, disp32(%RZP, %rcx, scale)
```

the upper 32 bits of `rcx` are cleared by the arithmetic operation on `ecx`. Note that any operation on `ecx` will clear the top 32 bits of `rcx`. This required masking operation can often be combined other useful operations. Note that this general form allows generation of addresses in a range of approximately 100GB, with the

[5]We have used the = operation to indicate assignment to the register on the left hand side. There are several instructions, such as `lea` or `movzx` that can be used to perform this assignment. Other instructions are written using ATT syntax.

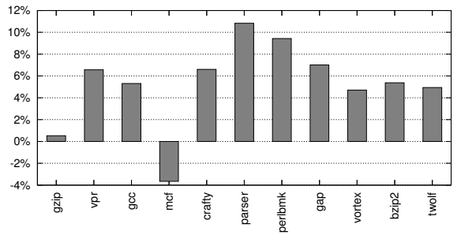



Figure 1: SPEC2000 SFI Performance Overhead for the ARM Cortex-A9.

valid 4GB range near the middle. By reserving and unmapping addresses outside the 4GB range we can ensure that any dereference of an address outside the valid range will lead to a fault. Clearly this scheme relies heavily on the very large 64-bit address space.

Finally, note that updates to the instruction pointer must align the address to 0 mod 32 and initialize the top 32-bits of address from RZP as in this example using `rdx`:

```
%edx = ...
and 0xfffffffe0, %edx
lea (%RZP, %rdx, 1), %rdx
jmp *%rdx
```

Our x86-64 SFI implementation is based on GCC 4.4.3, requiring a patch of about 2000 lines to the compiler, linker and assembler source. At a high level, the changes include supporting the new call/return sequences, making pointers and longs 32 bits, allocating `r15` for use as `RZB`, and constraining address generation to meet the above rules.

## 4 Evaluation

In this section we evaluate the performance of our ARM and x86-64 SFI schemes by comparing against the relevant non-SFI baselines, using C and benchmarks from SPEC2000 INT CPU [12]. Our main analysis is based on out-of-order CPUs, with additional measurements for in-order systems at the end of this section. The out-of-order systems we used for our experiments were:

- For x86-64, a 2.4GHz Intel Core 2 Quad with 8GB of RAM, running Ubuntu Linux 8.04, and
- For ARM, a 1GHz Cortex-A9 (Nvidia Tegra T20) with 512MB of RAM, running Ubuntu Linux 9.10.

### 4.1 ARM

For ARM, we compared LLVM 2.6 [13] to the same compiler modified to support our SFI scheme. Figure 1 summarizes the ARM results, with tabular data in Table 2. Average overhead is about 5% on the out-of-order

| | x86-64 SFI | SFI vs. -m32 | SFI vs. -m64 | ARM SFI |
|---|---|---|---|---|
| 164.gzip | 16.0 | 0.82 | 16.0 | 0.53 |
| 175.vpr | 1.60 | -5.06 | 1.60 | 6.57 |
| 176.gcc | 35.1 | 35.1 | 33.0 | 5.31 |
| 181.mcf | 1.34 | 1.34 | -42.6 | -3.65 |
| 186.crafty | 29.3 | -8.17 | 29.3 | 6.61 |
| 197.parser | -4.07 | -4.07 | -20.3 | 10.83 |
| 253.perlbmk | 34.6 | 26.6 | 34.6 | 9.43 |
| 254.gap | -4.46 | -4.46 | -5.09 | 7.01 |
| 255.vortex | 43.0 | 26.0 | 43.0 | 4.71 |
| 256.bzip2 | 21.6 | 4.84 | 21.6 | 5.38 |
| 300.twolf | 0.80 | -3.08 | 0.80 | 4.94 |
| geomean | 14.7 | 5.24 | 6.9 | 5.17 |

Table 2: SPEC2000 SFI Performance Overhead (percent). The first column compares x86-64 SFI overhead to the "oracle" baseline compiler.

| | ARM | ARM SFI | %inc. | %pad |
|---|---|---|---|---|
| 164.gzip | 73 | 90 | 24 | 13 |
| 175.vpr | 225 | 271 | 20 | 13 |
| 176.gcc | 1586 | 1931 | 22 | 14 |
| 181.mcf | 84 | 103 | 23 | 12 |
| 186.crafty | 320 | 384 | 20 | 12 |
| 197.parser | 219 | 265 | 21 | 12 |
| 253.perlbmk | 812 | 1009 | 24 | 14 |
| 254.gap | 531 | 636 | 20 | 11 |
| 255.vortex | 720 | 845 | 17 | 13 |
| 256.bzip2 | 74 | 92 | 24 | 13 |
| 300.twolf | 289 | 343 | 19 | 11 |

Table 3: ARM SPEC2000 text segment size in kilobytes, with % increase and % padding instructions.

Cortex-A9, and is fairly consistent across the benchmarks. Increases in binary size (Table 3) are comparable at around 20% (generally about 10% due to alignment padding and 10% due to added instructions, shown in the rightmost columns of the table). We believe the observed overhead comes primarily from the increase in code path length. For `mcf`, this benchmark is known to be data-cache intensive [17], a case in which the additional sandboxing instructions have minimal impact, and can sometimes be hidden by out-of-order execution on the Cortex-A9. We see the largest slowdowns for `gap`, `gzip`, and `perlbmk`. We suspect these overheads are a combination of increased path length and instruction cache penalties, although we do not have access to ARM hardware performance counter data to confirm this hypothesis.



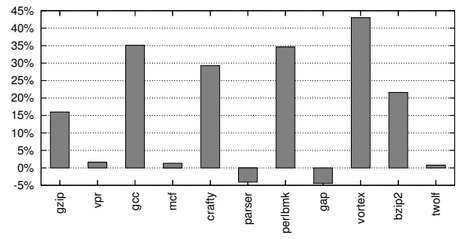



Figure 2: SPEC2000 SFI Performance Overhead for x86-64. SFI performance is compared to the faster of `-m32` and `-m64` compilation.

### 4.2 x86-64

Our x86-64 comparisons are based on GCC 4.4.3. The selection of a performance baseline is not straightforward. The available compilation modes for x86 are either 32-bit (ILP32, `-m32`) or 64-bit (LP64, `-m64`). Each represents a performance tradeoff, as demonstrated previously [15, 25]. In particular, the 32-bit compilation model's use of ILP32 base types means a smaller data working set compared to standard 64-bit compilation in GCC. On the other hand, use of the 64-bit instruction set offers additional registers and a more efficient register-based calling convention compared to standard 32-bit compilation. Ideally we would compare our SFI compiler to a version of GCC that uses ILP32 and the 64-bit instruction set, but without our SFI implementation. In the absence of such a compiler, we consider a hypothetical compiler that uses an oracle to automatically select the faster of `-m32` and `-m64` compilation. Unless otherwise noted all GCC compiles used the `-O2` optimization level.

Figure 2 and Table 2 provide x86-64 results, where average SFI overhead is about 5% compared to `-m32`, 7% compared to `-m64` and 15% compared to the oracle compiler. Across the benchmarks, the distribution is roughly bi-modal. For `parser` and `gap`, SFI performance is better than either `-m32` or `-m64` binaries (Table 4). These are also cases where `-m64` execution is slower than `-m32`, indicative of data-cache pressure, leading us to believe that the beneficial impact additional registers dominates SFI overhead. Three other benchmarks (`vpr`, `mcf` and `twolf`) show SFI impact is less than 2%. We believe these are memory-bound and do not benefit significantly from the additional registers.

At the other end of the range, four benchmarks, `gcc`, `crafty`, `perlbmk` and `vortex` show performance overhead greater than 25%. All run as fast or faster for `-m64` than `-m32`, suggesting that data-cache pressure does not dominate their performance. `Gcc`, `perlbmk` and `vortex` have large text, and we suspect SFI code-size increase may be contributing to instruction cache pressure. From hardware performance counter data, `crafty` shows a 26% increase in instructions retired and an increase in branch mispredicts from 2% to 8%, likely contributors to the observed SFI performance overhead. We have also observed that `perlbmk` and `vortex` are very sensitive to `memcpy` performance. Our x86-64 experiments are using a relative simple implementation of `memcpy`, to allow the same code to be used with and without the SFI sandbox. In our continuing work we are adapting a tuned `memcpy` implementation to work within our sandbox.

| | -m32 | -m64 | SFI |
|---|---|---|---|
| 164.gzip | 122 | 106 | 123 |
| 175.vpr | 87 | 81.3 | 82.6 |
| 176.gcc | 47.3 | 48.0 | 63.9 |
| 181.mcf | 59.5 | 105 | 60.3 |
| 186.crafty | 60 | 42.6 | 55.1 |
| 197.parser | 123 | 148 | 118 |
| 253.perlbmk | 86.9 | 81.7 | 110 |
| 254.gap | 60.5 | 60.9 | 57.8 |
| 255.vortex | 99.2 | 87.4 | 125 |
| 256.bzip2 | 99.2 | 85.5 | 104 |
| 300.twolf | 130 | 125 | 126 |

Table 4: SPEC2000 x86-64 execution times, in seconds.

| | -m32 | -m64 | SFI |
|---|---|---|---|
| 164.gzip | 82 | 85 | 155 |
| 175.vpr | 239 | 244 | 350 |
| 176.gcc | 1868 | 2057 | 3452 |
| 181.mcf | 20 | 23 | 33 |
| 186.crafty | 286 | 257 | 395 |
| 197.parser | 243 | 265 | 510 |
| 253.perlbmk | 746 | 835 | 1404 |
| 254.gap | 955 | 1015 | 1641 |
| 255.vortex | 643 | 620 | 993 |
| 256.bzip2 | 98 | 95 | 159 |
| 300.twolf | 375 | 410 | 617 |

Table 5: SPEC2000 x86 text sizes, in kilobytes.

### 4.3 In-Order vs. Out-of-Order CPUs

We suspected that the overhead of our SFI scheme would be hidden in part by CPU microarchitectures that better exploit instruction-level parallelism. In particular, we suspected we would be helped by the ability of out-of-order CPUs to schedule around any bottlenecks that SFI introduces. Fortunately, both architectures we tested have multiple implementations, including recent products with in-order dispatch. To test our hypothesis, we ran a subset of our benchmarks on in-order machines:

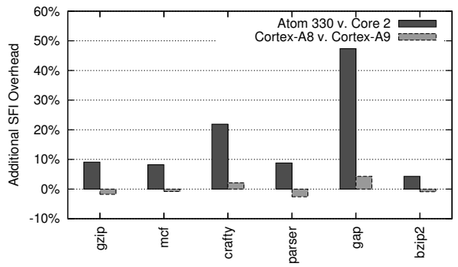



Figure 3: Additional SPEC2000 SFI overhead on in-order microarchitectures.

| | Core 2 | Atom 330 | A9 | A8 |
|---|---|---|---|---|
| 164.gzip | 16.0 | 25.1 | 4.4 | 2.6 |
| 181.mcf | -42.6 | -34.4 | -0.2 | -1.0 |
| 186.crafty | 29.3 | 51.2 | 4.2 | 6.3 |
| 197.parser | -20.3 | -11.5 | 3.2 | 0.6 |
| 254.gap | -5.09 | 42.3 | 3.4 | 7.7 |
| 256.bzip2 | 21.6 | 25.9 | 2.9 | 2.0 |
| geomean | 6.89 | 18.5 | 3.0 | 3.0 |

Table 6: Comparison of SPEC2000 overhead (percent) for in-order vs. out-of-order microarchitecture.

- A 1.6GHz Intel Atom 330 with 2GB of RAM, running Ubuntu Linux 9.10.
- A 500MHz Cortex-A8 (Texas Instruments OMAP3540) with 256MB of RAM, running Ångström Linux.

The results are shown in Figure 3 and Table 6. For our x86-64 SFI scheme, the incremental overhead can be significantly higher on the Atom 330 compared to a Core 2 Duo. This suggests out-of-order execution can help hide the overhead of SFI, although other factors may also contribute, including much smaller caches on the Atom part and the fact that GCC's 64-bit code generation may be biased towards the Core 2 microarchitecture. These results should be considered preliminary, as there are a number of optimizations for Atom that are not yet available in our compiler, including Atom-specific instruction scheduling and better selection of no-ops. Generation of efficient SFI code for in-order x86-64 systems is an area of continuing work.

The story on ARM is different. While some benchmarks (notably `gap`) have higher overhead, some (such as `parser`) have equally reduced overhead. We were surprised by this result, and suggest two factors to account for it. First, microarchitectural evaluation of the Cortex-A8 [3] suggests that the instruction sequences produced by our SFI can be issued without encountering a hazard that would cause a pipeline stall. Second, we suggest that the Cortex-A9, as the first widely-available out-of-order ARM chip, might not match the maturity and sophistication of the Core 2 Quad.

## 5 Discussion

Given our initial goal to impact execution time by less than 10%, we believe these SFI designs are promising. At this level of performance, most developers targeting our system would do better to tune their own code rather than worry about SFI overhead. At the same time, the geometric mean commonly used to report SPEC results does a poor job of capturing the system's performance characteristics; nobody should expect to get "average" performance. As such we will continue our efforts to reduce the impact of SFI for the cases with the largest slowdowns.

Our work fulfills a prediction that the costs of SFI would become lower over time [28]. While thoughtful design has certainly helped minimize SFI performance impact, our experiments also suggest how SFI has benefited from trends in microarchitecture. Out-of-order execution, multi-issue architectures, and the effective gap between memory speed and CPU speed all contribute to reduce the impact of the register-register instructions used by our sandboxing schemes.

We were surprised by the low overhead of the ARM sandbox, and that the x86-64 sandbox overhead should be so much larger by comparison. Clever ARM instruction encodings definitely contributed. Our design directly benefits from the ARM's powerful bit-clear instruction and from predication on stores. It usually requires one instruction per sandboxed ARM operation, whereas the x86-64 sandbox frequently requires extra instructions for address calculations and adds a prefix byte to many instructions. The regularity of the ARM instruction set and smaller bundles (16 vs. 32 bytes) also means that less padding is required for the ARM, hence less instruction cache pressure. The x86-64 design also induces branch misprediction through our omission of the `ret` instruction. By comparison the ARM design uses the normal return idiom hence minimal impact on branch prediction. We also note that the x86-64 systems are generally clocked at a much higher rate than the ARM systems, making the relative distance to memory a possible factor. Unfortunately we do not have data to explore this question thoroughly at this time.

We were initially troubled by the result that our system improves performance for so many benchmarks compared to the common `-m32` compilation mode. This clearly results from the ability of our system to leverage features of the 64-bit instruction set. There is a sense in which the comparison is unfair, as running a 32-bit binary on a 64-bit machine leaves a lot of resources idle. Our results demonstrate in part the benefit of exploiting those additional resources.

We were also surprised by the magnitude of the positive impact of ILP32 primitive types for a 64-bit binary. For now our x86-64 design benefits from this as yet unexploited opportunity, although based on our experience the community might do well to consider making ILP32 a standard option for x86-64 execution.

In our continuing work we are pursuing opportunities to reduce SFI overhead of our x86-64 system, which we do not consider satisfactory. Our current alignment implementation is conservative, and we have identified a number of opportunities to reduce related padding. We will be moving to GCC version 4.5 which has instruction-scheduling improvements for in-order Atom systems. In the fullness of time we look forward to developing an infrastructure for profile-guided optimization, which should provide opportunities for both instruction cache and branch optimizations.

## 6 Related Work

Our work draws directly on Native Client, a previous system for sandboxing 32-bit x86 modules [30]. Our scheme for optimizing stack references was informed by an earlier system described by McCamant and Morrisett [18]. We were heavily influenced by the original software fault isolation work by Wahbe, Lucco, Anderson and Graham [28].

Although there is a large body of published research on software fault isolation, we are aware of no publications that specifically explore SFI for ARM or for the 64-bit extensions of the x86 instruction set. SFI for SPARC may be the most thoroughly studied, being the subject of the original SFI paper by Wahbe et al. [28] and numerous subsequent studies by collaborators of Wahbe and Lucco [2, 16, 11] and independent investigators [4, 5, 8, 9, 10, 14, 22, 29]. As this work matured, much of the community's attention turned to a more virtual machine-oriented approach to isolation, incorporating a trusted compiler or interpreter into the trusted core of the system.

The ubiquity of the 32-bit x86 instruction set has catalyzed development of a number of additional sandboxing schemes. MiSFIT [23] contemplated use of software fault isolation to constrain untrusted kernel modules [24]. Unlike our system, they relied on a trusted compiler rather than a validator. SystemTAP and XFI [21, 7] further contemplate x86 sandboxing schemes for kernel extension modules. McCamant and Morrisett [18, 19] studied x86 SFI towards the goals of system security and reducing the performance impact of SFI.

Compared to our sandboxing schemes, CFI [1] provides finer-grained control flow integrity. Whereas our systems only guarantee indirect control flow will target an aligned address in the text segment, CFI can restrict a specific control transfer to a fairly arbitrary subset of known targets. While this more precise control is useful in some scenarios, such as ensuring integrity of translations from higher-level languages, our use of alignment constraints helps simplify our design and implementation. CFI also has somewhat higher average overhead (15% on SPEC2000), not surprising since its instrumentation sequences are longer than ours. XFI [7] adds to CFI further integrity constraints such as on memory and the stack, with additional overhead. More recently, BGI [6] considers an innovative scheme for constraining the memory activity of device drivers, using a large bitmap to track memory accessibility at very fine granularity. None of these projects considered the problem of operating system portability, a key requirement of our systems.

The Nooks system [26] enhances operating system kernel reliability by isolating trusted kernel code from untrusted device driver modules using a transparent OS layer called the Nooks Isolation Manager (NIM). Like Native Client, NIM uses memory protection to isolate untrusted modules. As the NIM operates in the kernel, x86 segments are not available. The NIM instead uses a private page table for each extension module. To change protection domains, the NIM updates the x86 page table base address, an operation that has the side effect of flushing the x86 Translation Lookaside Buffer (TLB). In this way, NIM's use of page tables suggests an alternative to segment protection as used by NaCl-x86-32. While a performance analysis of these two approaches would likely expose interesting differences, the comparison is moot on the x86 as one mechanism is available only within the kernel and the other only outside the kernel. A critical distinction between Nooks and our sandboxing schemes is that Nooks is designed only to protect against unintentional bugs, not abuse. In contrast, our sandboxing schemes must be resistant to attempted deliberate abuse, mandating our mechanisms for reliable x86 disassembly and control flow integrity. These mechanisms have no analog in Nooks.

Our system uses a static validator rather than a trusted compiler, similar to validators described for other systems [7, 18, 19, 21], applying the concept of proof-carrying code [20]. This has the benefit of greatly reducing the size of the trusted computing base [27], and obviates the need for cryptographic signatures from the compiler. Apart from simplifying the security implementation, this has the further benefit of opening our system to 3rd-party tool chains.

## 7 Conclusion

This paper has presented practical software fault isolation systems for ARM and for 64-bit x86. We believe

these systems demonstrate that the performance overhead of SFI on modern CPU implementations is small enough to make it a practical option for general purpose use when executing untrusted native code. Our experience indicates that SFI benefits from trends in microarchitecture, such as out-of-order and multi-issue CPU cores, although further optimization may be required to avoid penalties on some recent low power in-order cores. We further found that for data-bound workloads, memory latency can hide the impact of SFI.

Source code for Google Native Client can be found at: `http://code.google.com/p/nativeclient/`.

## References


[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[2] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. *SIGPLAN Not.*, 31(5):127–136, 1996.

[3] ARM Limited. Cortex A8 technical reference manual. `http://infocenter.arm.com/help/index.jsp?topic=com.arm.doc.ddi0344/index.html`, 2006.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[5] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[6] M. Castro, M. Costa, J. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *2009 Symposium on Operating System Principles*, pages 45–58, October 2009.

[7] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address spaces. In *OSDI '06: 7th Symposium on Operating Systems Design And Implementation*, pages 75–88, November 2006.

[8] B. Ford. VXA: A virtual architecture for durable compressed archives. In *USENIX File and Storage Technologies*, December 2005.

[9] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *2008 USENIX Annual Technical Conference*, June 2008.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[11] S. Graham, S. Lucco, and R. Wahbe. Adaptable binary programs. In *Proceedings of the 1995 USENIX Technical Conference*, 1995.

[12] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[13] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Masters Thesis, Computer Science Department, University of Illinois, 2003.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999.

[15] J. Liu and Y. Wu. Performance characterization of the 64-bit x86 architecture from compiler optimizations' perspective. In *Proceedings of the International Conference on Compiler Construction, CC'06*, 2006.

[16] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A universal substrate for web programming. In *Fourth International World Wide Web Conference*, 1995.

[17] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn. Profile-guided post-link stride prefetching. In *Proceedings of the ACM International Conference on Supercomputing, ICS'02*, 2002.

[18] S. McCamant and G. Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical Report MIT-CSAIL-TR-2005-030, MIT Computer Science and Artificial Intelligence Laboratory, 2005.

[19] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, August 2006.

[20] G. Necula. Proof carrying code. In *Principles of Programming Languages*, 1997.

[21] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and J. Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*, pages 49–64, July 2005.

[22] J. Richter. *CLR via C#, Second Edition*. Microsoft Press, 2006.

[23] C. Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, June 1997.

[24] C. Small and M. Seltzer. VINO: An integrated platform for operating systems and database research. Technical Report TR-30-94, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA, 1994.

[25] Sun Microsystems. Compressed OOPs in the HotSpot JVM. `http://wikis.sun.com/display/HotSpotInternals/CompressedOops`.

[26] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering device drivers. In *6th USENIX Symposium on Operating Systems Design and Implementation*, December 2004.

[27] U. S. Department of Defense, Computer Security Center. Trusted computer system evaluation criteria, December 1985.

[28] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[29] C. Waldspurger. Memory resource management in VMware ESX Server. In *5th Symposium on Operating Systems Design and Implementation*, December 2002.

[30] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.

# Making Linux Protection Mechanisms Egalitarian with UserFS

Taesoo Kim and Nickolai Zeldovich
*MIT CSAIL*

## ABSTRACT

UserFS provides egalitarian OS protection mechanisms in Linux. UserFS allows any user—not just the system administrator—to allocate Unix user IDs, to use `chroot`, and to set up firewall rules in order to confine untrusted code. One key idea in UserFS is representing user IDs as files in a `/proc`-like file system, thus allowing applications to manage user IDs like any other files, by setting permissions and passing file descriptors over Unix domain sockets. UserFS addresses several challenges in making user IDs egalitarian, including accountability, resource allocation, persistence, and UID reuse. We have ported several applications to take advantage of UserFS; by changing just tens to hundreds of lines of code, we prevented attackers from exploiting application-level vulnerabilities, such as code injection or missing ACL checks in a PHP-based wiki application. Implementing UserFS requires minimal changes to the Linux kernel—a single 3,000-line kernel module—and incurs no performance overhead for most operations, making it practical to deploy on real systems.

## 1 INTRODUCTION

OS protection mechanisms are key to mediating access to OS-managed resources, such as the file system, the network, or other physical devices. For example, system administrators can use Unix user IDs to ensure that different users cannot corrupt each other's files; they can set up a `chroot` jail to prevent a web server from accessing unrelated files; or they can create firewall rules to control network access to their machine. Most operating systems provide a range of such mechanisms that help administrators enforce their security policies.

While these protection mechanisms can enforce the administrator's policy, many applications have their own security policies for OS-managed resources. For instance, an email client may want to execute suspicious attachments in isolation, without access to the user's files; a networked game may want to configure a firewall to make sure it does not receive unwanted network traffic that may exploit a vulnerability; and a web browser may want to precisely control what files and devices (such as a video camera) different sites or plugins can access. Unfortunately, typical OS protection mechanisms are only accessible to the administrator: an ordinary Unix user cannot allocate a new user ID, use `chroot`, or change firewall rules, forcing applications to invent their own protection techniques like system call interposition [15], binary rewriting [30] or analysis [13, 45], or interposing on system accesses in a language runtime like Javascript.

This paper presents the design of UserFS, a kernel framework that allows any application to use traditional OS protection mechanisms on a Unix system, and a prototype implementation of UserFS for Linux. UserFS makes protection mechanisms *egalitarian*, so that any user—not just the system administrator—can allocate new user IDs, set up firewall rules, and isolate processes using `chroot`. By using the operating system's own protection mechanisms, applications can avoid race conditions and ambiguities associated with system call interposition [14, 43], can confine existing code without having to recompile or rewrite it in a new language, and can enforce a coherent security policy for large applications that might span several runtime environments, such as both Javascript and Native Client [45], or Java and JNI code.

Allowing arbitrary users to manipulate OS protection mechanisms through UserFS requires addressing several challenges. First, UserFS must ensure that a malicious user cannot exploit these mechanisms to violate another application's security policy, perhaps by re-using a previously allocated user ID, or by running setuid-root programs in a malicious `chroot` environment. Second, user IDs are often used in Unix for accountability and auditing, and UserFS must ensure that a system administrator can attribute actions to users that he or she knows about, even for processes that are running with a newly-allocated user ID. Finally, UserFS should to be compatible with existing applications, interfaces, and kernel components whenever possible, to make it easy to incrementally deploy UserFS in practical systems.

UserFS addresses these challenges with a few key ideas. First, UserFS allows applications to allocate user IDs that are indistinguishable from traditional user IDs managed by the system administrator. This ensures that existing applications do not need to be modified to support application-allocated protection domains, and that existing UID-based protection mechanisms like file permissions can be reused. Second, UserFS maintains a shadow generation number associated with each user ID, to make sure that setuid executables for a given UID cannot be used to obtain privileges once the UID has been reused by a new application. Third, UserFS represents allocated user

IDs using files in a special file system. This makes it easy to manipulate user IDs, much like using the `/proc` file system on Linux, and applications can use file descriptor passing to delegate privileges and implement authentication logic. Finally, UserFS uses information about what user ID allocated what other user IDs to determine what setuid executables can be trusted in any given `chroot` environment, as will be described later.

We have implemented a prototype of UserFS for Linux purely as a kernel module, consisting of less than 3,000 lines of code, along with user-level support libraries for C and PHP-based applications. UserFS imposes no performance overhead for most existing operations, and only performs an additional check when running setuid executables. We modified several applications to enforce security policies using UserFS, including Google's Chromium web browser, a PHP-based wiki application, an FTP server, `ssh-agent`, and Unix commands like `bash` and `su`, all with minimal code modifications, suggesting that UserFS is easy to use. We further show that our modified wiki is not vulnerable by design to 5 out of 6 security vulnerabilities found in that application over the past several years.

The key contribution of this work is the first system that allows Linux protection and isolation mechanisms to be freely used by non-root code. This improves overall security both by allowing applications to enforce their policies in the OS, and by reducing the amount of code that needs to run as root in the first place (for example to set up `chroot` jails, create new user accounts, or configure firewall rules).

The rest of this paper is structured as follows. Section 2 provides more concrete examples of applications that would benefit from access to OS protection mechanisms. Section 3 describes the design of UserFS in more detail, and Section 4 covers our prototype implementation. We illustrate how we modified existing applications to take advantage of UserFS in Section 5, and Section 6 evaluates the security and performance of UserFS. Section 7 surveys related work, Section 8 discusses the limitations of our system, and Section 9 concludes.

## 2 Motivation and goals

The main goal of UserFS is to help applications reduce the amount of trusted code, by allowing them to use traditionally privileged OS protection mechanisms to control access to system resources, such as the file system and the network. We believe this will allow many applications to improve their security, by preventing compromises where an attacker takes advantage of an application's excessive OS-level privileges. However, UserFS is not a security panacea, and programmers will still need to think about a wide range of other security issues from cryptography to cross-site scripting attacks. The rest of this section provides several motivating examples in which UserFS can improve security.

**Avoiding root privileges in existing applications.** Typical Unix systems run a large amount of code as root in order to perform privileged operations. For example, network services that allow user login, such as an FTP server, `sshd`, or an IMAP server often run as root in order to authenticate users and invoke `setuid()` to acquire their privileges on login. Unfortunately, these same network services are the parts of the system most exposed to attack from external adversaries, making any bug in their code a potential security vulnerability. While some attempts have been made to privilege-separate network services, such as with OpenSSH [39], it requires carefully re-designing the application and explicitly moving state between privileged and unprivileged components. By allowing processes to explicitly manipulate Unix users as file descriptors, and pass them between processes, UserFS eliminates the need to run network services as the root user, as we will show in Section 5.3.

In addition to network services, users themselves often want to run code as root, in order to perform currently-privileged operations. For instance, `chroot` can be useful in building a complex software package that has many dependencies, but unfortunately `chroot` can only be invoked by root. By allowing users to use a range of mechanisms currently reserved for the system administrator, UserFS further reduces the need to run code as root.

**Sandboxing untrusted code.** Users often interact with untrusted or partially-trusted code or data on their computers. For example, users may receive attachments via email, or download untrusted files from the web. Opening or executing these files may exploit vulnerabilities in the user's system. While it's possible for the mail client or web browser to handle a few types of attachments (such as HTML files) safely, in the general case opening the document will require running a wide range of existing applications (e.g. OpenOffice for Word files, or Adobe Acrobat to view PDFs). These helper applications, even if they are not malicious themselves, might perform undesirable actions when viewing malicious documents, such as a Word macro virus or a PDF file that exploits a buffer overflow in Acrobat.

Guarding against these problems requires isolating the suspect application from the rest of the system, while providing a limited degree of sharing (such as initializing Acrobat with the user's preferences). With UserFS, the mail client or web browser can allocate a fresh user ID to view a suspicious file, and use firewall rules to ensure the application does not abuse the user's network connection (e.g. to send spam), and Section 5.2 will describe how UserFS helps Unix users isolate partially-trusted or untrusted applications in this manner.



**Enforcing separation in privilege-separated applications.** One approach to building high-security applications is to follow the principle of least privilege [40] by breaking up an application into several components, each of which has the minimum privileges necessary. For instance, OpenSSH [39], qmail [3], and the Chromium browser [2] follow this model, and tools exist to help programmers privilege-separate existing applications [7]. One problem is that executing components with less privileges requires either root privilege to start with (and applications that are not fully-trusted to start with are unlikely to have root privileges), or other complex mechanisms. With UserFS, privilege-separated applications can use existing OS protection primitives to enforce isolation between their components, without requiring root privileges to do so. We hope that, by making it easier to execute code with less privileges, UserFS encourages more applications to improve their security by reducing privileges and running as multiple components. As an example, Section 5.4 shows how UserFS can isolate different processes in the Chromium web browser.

**Exporting OS resources in higher-level runtimes.** Finally, there are many higher-level runtimes running on a typical desktop system, such as Javascript, Flash, Native Client [45], and Java. Applications running on top of these runtimes often want to access underlying OS resources, including the file system, the network, and local devices such as a video camera. This currently forces the runtimes to implement their own protection schemes, e.g. based on file names, which can be fragile, and worse yet, enforce different policies depending on what runtime an application happens to use. By using UserFS, runtimes can delegate enforcement of security checks to the OS kernel, by allocating a fresh user ID for logical protection domains managed by the runtime. For example, Section 5.1 shows how UserFS can enforce security policies for a PHP web application. In the future, we hope the same mechanisms can be used to implement a coherent security policy for one application across all runtimes that it might use.

## 3 Kernel interface design

To help applications reduce the amount of trusted code, UserFS allows any application to *allocate new principals*; in Unix, principals are user IDs and group IDs. An application can then enforce its desired security policy by first allocating new principals for its different components, then, second, setting file permissions—i.e., read, write, and execute privileges for principals—to match its security policy, and finally, running its different components under the newly-allocated principals.

A slight complication arises from the fact that, in many Unix systems, there are a wide range of resources available to all applications by default, such as the `/tmp` directory or the network stack. Thus, to restrict untrusted code from accessing resources that are accessible by default, UserFS also allows applications to *impose restrictions* on a process, in the form of `chroot` jails or firewall rules. The rest of this section describes the design of the UserFS kernel mechanisms that provide these features.

### 3.1 User ID allocation

The first function of UserFS is to allow any application to allocate a new principal, in the form of a Unix user ID. At a naïvely high level, allocating user IDs is easy: pick a previously unused user ID value and return it to the application. However, there are four technical challenges that must be addressed in practice:

- When is it safe for a process to exercise the privileges of another user ID, or to change to a different UID? Traditional Unix provides two extremes, neither of which are sufficient for our requirements: non-root processes can only exercise the privileges of their current UID, and root processes can exercise everyone's privileges.
- How do we keep track of the resources associated with user IDs? Traditional Unix systems largely rely on UIDs to attribute processes to users, to implement auditing, and to perform resource accounting, but if users are able to create new user IDs, they may be able to evade UID-based accounting mechanisms.
- How do we recycle user ID values? Most Unix systems and applications reserve 32 bits of space for user ID values, and an adversary or a busy system can quickly exhaust $2^{32}$ user ID values. On the other hand, if we recycle UIDs, we must make sure that the previous owner of a particular UID cannot obtain privileges over the new owner of the same UID value.
- Finally, how do we keep user ID allocations persistent across reboots of the kernel?

We will now describe how UserFS addresses these challenges, in turn.

#### 3.1.1 Representing privileges

UserFS represents user IDs with *files* that we will call *Ufiles* in a special `/proc`-like file system that, by convention, is mounted as `/userfs`. Privileges with respect to a specific user ID can thus be represented by *file descriptors* pointing to the appropriate Ufile. Any process that has an open file descriptor corresponding to a Ufile can issue a `USERFS_IOC_SETUID` *ioctl* on that file descriptor to change the process's current UID (more specifically, euid) to the Ufile's UID.

Aside from the special *ioctl* calls, file descriptors for Ufiles behave exactly like any other Unix file descriptor. For instance, an application can keep multiple file descriptors for different user IDs open at the same time, and switch its process UID back and forth between them. Applications can also use file descriptor passing over Unix domain sockets to pass privileges between processes. This can be useful in implementing user authentication or login, by allowing an authentication daemon to accept login requests over a Unix domain socket, and to return a file descriptor for that user's Ufile if the supplied credential (e.g. password) was correct.

Finally, each Ufile under `/userfs` has an owner user and group associated with it, along with user and group permissions. These permissions control what other users and groups can obtain the privileges of a particular UID by opening its via path name. By default, a Ufile is owned by the user and group IDs of the process that initially allocated that UID, and has Unix permissions 600 (i.e. accessible by owner, but not by group or others), allowing the process that allocated the UID to access it initially. A process can always access the Ufile for the process's current UID, regardless of the permissions on that Ufile (this allows a process to always obtain a file descriptor for its current UID and pass it to others via FD passing).

### 3.1.2 Accountability hierarchy

Ufiles help represent privileges over a particular user ID, but to provide accountability, our system must also be able to say what user is responsible for a particular user ID. This is useful for accounting and auditing purposes: tracking what users are using disk space, running CPU-intensive processes, or allocating many user IDs via UserFS, or tracking down what user tried to exploit some vulnerability a week ago.

To provide accountability, UserFS implements a hierarchy of user IDs. In particular, each UID has a parent UID associated with it. The parent UID of existing Unix users is root (0), including the parent of root itself. For dynamically-allocated user IDs, the parent is the user ID of the process that allocated that UID (which in turn has its own parent UID). UserFS represents this UID hierarchy with directories under `/userfs`, as illustrated in Figure 1. For convenience, UserFS also provides symbolic links for each UID under `/userfs` that point to the hierarchical name of that UID, which helps the system administrator figure out who is responsible for a particular UID.

In addition to the `USERFS_IOC_SETUID` ioctl that was mentioned earlier, UserFS supports three more operations. First, a process can allocate new UIDs by issuing a `USERFS_IOC_ALLOC` ioctl on a Ufile. This allocates a new UID as a child of the Ufile's UID, and the value of the newly allocated UID is returned as the result of the ioctl. A process can also de-allocate UIDs by performing an `rmdir` on the appropriate directory under `/userfs`. This will recursively de-allocate that UID and all of its child UIDs (i.e. it will work even on non-empty directories), and kill any processes running under those UIDs, for reasons we will describe shortly. Finally, a process can move a UID in the hierarchy using `rename` (for example, if one user is no longer interested in being responsible for a particular UID, but another user is willing to provide resources for it).

Finally, accountability information may be important long after the UID in question has been de-allocated (e.g. the administrator wants to know who was responsible for a break-in attempt, but the UID in the log associated with the attempt has been de-allocated already). To address this problem, UserFS uses syslog to log all allocations, so that an administrator can reconstruct who was responsible for that UID at any point in time.

### 3.1.3 UID reuse

An ideal system would provide a unique identifier to every principal that ever existed. Unfortunately, most Unix kernel data structures and applications only allocate space for a 32-bit user ID value, and an adversary can easily force a system to allocate $2^{32}$ user IDs. To solve this problem, UserFS associates a 64-bit *generation number* with every allocated UID[1], in order to distinguish between two principals that happen to have had the same 32-bit UID value at different times. The kernel ensures that generation numbers are unique by always incrementing the generation number when the UID is deallocated. However, as we just mentioned, there isn't enough space to store the generation number along with the user ID in every kernel data structure. UserFS deals with this on a case-by-case basis:

**Processes.** UserFS assumes that the current UID of a process always corresponds to the latest generation number for that UID. This is enforced by killing every process whose current UID has been deallocated.

**Open Ufiles.** UserFS keeps track of the generation number for each open file descriptor of a Ufile, and verifies that the generation number is current before proceeding with any ioctl on that file descriptor (such as `USERFS_IOC_SETUID`). Once a UID has been reused, the current UID generation number is incremented, and leftover file descriptors for the old Ufile will be unusable. This ensures that a process that had privileges over a UID in the past cannot exercise those privileges once the UID is reused.

[1] It would take an attacker thousands of years to allocate $2^{64}$ UIDs, even at a rate of 1 million UIDs per second.



| Path name | Role |
|---|---|
| `/userfs/ctl` | Ufile for root (UID 0). |
| `/userfs/1001/ctl` | Ufile for user 1001 (parent UID 0). |
| `/userfs/1001/5001/ctl` | Ufile for user 5001 (allocated by parent UID 1001). |
| `/userfs/1001/5001/5002/ctl` | Ufile for user 5003 (allocated by parent UID 5001). |
| `/userfs/1001/5003/ctl` | Ufile for user 5003 (allocated by parent UID 1001). |
| `/userfs/1002/ctl` | Ufile for user 1002 (parent UID 0). |
| `/userfs/5001` | Symbolic link to `1001/5001`. |
| `/userfs/5002` | Symbolic link to `1001/5001/5002`. |
| `/userfs/5003` | Symbolic link to `1001/5003`. |

**Figure 1**: An overview of the files exported via UserFS in a system with two traditional Unix accounts (UID 1001 and 1002), and three dynamically-allocated accounts (5001, 5002, and 5003). Not shown are system UIDs that would likely be present on any system (users such as `bin`, `nobody`, etc), or directories that are implied by the `ctl` files. Each `ctl` file supports two *ioctl*s: `USERFS_IOC_SETUID` and `USERFS_IOC_ALLOC`.

**Setuid files.** Setuid files are similar to a file descriptor for a Ufile, in the sense that they can be used to gain the privileges of a UID. To prevent a stale setuid file from being used to start a process with the same UID in the future, UserFS keeps track of the file owner's UID generation number for every setuid file in that file's extended attributes. (Extended attributes are supported by many file systems, including ext2, ext3, and ext4. Moreover, small extended attributes, such as our generation number, are often stored in the inode itself, avoiding additional seeks in the common case.) UserFS sets the generation number attribute when the file is marked setuid, or when its owner changes, and checks whether the generation number is still current when the setuid file is executed.

**Non-setuid files, directories, and other resources.** UserFS does not keep track of generation numbers for the UID owners of files, directories, system V semaphores, and so on. The assumption is that it's the previous UID owner's responsibility to get rid of any data or resources they do not want to be accessed by the next process that gets the same UID value. This is potentially risky, if sensitive data has been left on disk by some process, but is the best we have been able to do without changing large parts of the kernel.

There are several ways of addressing the problem of leftover files, which may be adopted in the future. First, the on-disk inode could be changed to keep track of the generation number along with the UID for each file. This approach would require significant changes to the kernel and file system, and would impose a minor runtime performance overhead for all file accesses. Second, the file system could be scanned to find orphaned files, much in the same way that UserFS scans the process table to kill processes running with a deallocated UID. This approach would make user deallocation expensive, although it would not require modifying the file system itself. Finally, each application could run sensitive processes with write access to only a limited set of directories, which can be garbage-collected by the application when it deletes the UID. Since none of the approaches are fully satisfactory, our design leaves the problem to the application, out of concern that imposing any performance overheads or extensive kernel changes would preclude the use of UserFS altogether.

### 3.1.4 Persistence

UserFS must maintain two pieces of persistent state. First, UserFS must make sure that generation numbers are not reused across reboot; otherwise an attacker could use a setuid file to gain another application's privileges when a UID is reused with the same generation number. One way to achieve this would be to keep track of the last generation number for each UID; however this would be costly to store. Instead, UserFS maintains generation numbers only for allocated UIDs, and just one "next" generation number representing all un-allocated UIDs. UserFS increments this next generation number when any UID is allocated or deallocated, and uses its current value when a new UID is allocated. To ensure that generation numbers are not reused in the case of a system crash, UserFS synchronously increments the next generation number on disk. As an important optimization, UserFS batches on-disk increments in groups of 1,000 (i.e., it only update the on-disk next generation number after 1,000 increments), and it always increments the next generation counter by 1,000 on startup to account for possibly-lost increments.

Second, UserFS must allow applications to keep using the same dynamically-allocated UIDs after reboot (e.g. if the file system contains data and/or setuid files owned by that UID). This involves keeping track of the generation number and parent UID for every allocated UID, as well as the owner UID and GID for the corresponding Ufile. UserFS maintains a list of such records in a file (`/etc/userfs_uid`), as shown in Figure 2. The permissions for the Ufile are stored as part of the owner value (if the owner UID or GID is zero, the corresponding permissions are 0, and if the owner UID or GID is non-zero, the corresponding permissions are read-write). The generation numbers of the parent UID, owner UID, and owner

GID are not tracked; the parent UID is necessarily current (otherwise this child would have been deallocated), and the owner UID and GID are left up to the Ufile owner.

UserFS lazily updates this on-disk data structure; deletion is implemented in-place by setting the UID value to $-1$. If an application wants to rely on the Ufile being present after reboot, it can force that Ufile's persistent record to be written to disk by issuing an `fsync` on the Ufile's file descriptor.

As an optimization, UserFS also allows non-persistent UIDs to be allocated (for isolating processes that do not store any persistent data in the file system under their UID). To implement this, the `USERFS_IOC_ALLOC` ioctl takes one argument that indicates whether the new UID should be persistent or not; persistent UIDs can only be allocated to persistent parents.

As a practical matter, UserFS partitions the 32-bit UID space into UIDs reserved for system use (0 through $2^{30}-1$), persistent dynamically-allocated UIDs ($2^{30}$ through $2^{31}-1$), non-persistent dynamically-allocated UIDs ($2^{31}$ through $2^{31}+2^{30}-1$), and more reserved UIDs ($2^{31}+2^{30}$ through $2^{32}-1$). This makes it easy to determine whether a particular UID is persistent, and avoids conflicts with most system-allocated UIDs at either end of the UID number space. UserFS provides modified `adduser` and `deluser` programs that create and delete Ufiles when they add or remove users from the system (to allow those users to allocate new UIDs via ioctls on their Ufile), and assumes that the system administrator will not use UIDs in the dynamically-allocated range.

### 3.2 Restriction mechanisms

To prevent malicious code from accessing resources that are accessible to everyone by default (such as `/tmp` or the network), UserFS allows applications to take advantage of existing restriction mechanisms: `chroot` to limit access to the file system namespace, and firewall rules to limit access to the network.

#### 3.2.1 *File system namespaces*

To prevent processes from accessing files that are accessible by default, UserFS allows any user to invoke `chroot`. There are two potential problems associated with this: setuid programs that will behave incorrectly in a `chroot` environment, and arbitrary programs attempting to escape from a `chroot` jail by recursive use of `chroot` itself.

**Setuid programs.** If a setuid program runs in a `chroot` environment, it can behave in unpredictable ways—for instance, a setuid-root `su` program may read a user-supplied `/etc/passwd` file and grant the caller root access because it assumed that root's password in its version of `/etc/passwd` was authentic. UserFS relies on the user ID hierarchy to address this problem. In particular, after user $U$ calls `chroot`, UserFS will only honor setuid bits for files owned by UIDs that are descendants of $U$. In the corner case of root invoking `chroot`, every user is a descendant of root, and thus every setuid program will still be honored, as on a regular Linux system.

UserFS only keeps track of the last UID to call `chroot` for a given process (inherited across fork). If one user performs `chroot` inside a second user's jail, it is the responsibility of the first user to verify that it's creating a `chroot` environment acceptable to all of its descendants. In practice, we expect that the first user will be a descendant of the second user (because he is executing inside the second user's jail), so this requirement will not pose significant problems.

**Escaping chroot.** The Linux `chroot` mechanism works by effectively maintaining a single "barrier" at the specified root directory that prevents the process from evaluating `..` (parent directory) of that process's root directory. A process can escape a `chroot` jail by obtaining a reference (either a file descriptor or current working directory) to a directory outside the `chroot`'ed hierarchy, and using that reference to walk up the `..` pointers to the true file system root. Even if an application properly uses `chroot` to confine a process, the kernel only keeps track of one root directory pointer per process, so a malicious process in a `chroot` jail could confine itself to a second `chroot` jail while maintaining a handle on a directory outside this second jail, and use that handle to escape both jails.

To prevent this problem, UserFS enforces three rules for `chroot` invoked by non-root users. First, to ensure a process cannot maintain a current working directory outside the `chroot` environment, UserFS requires that `chroot` callers set their directory to the `chroot` target directory ahead of time. Second, UserFS checks that a process calling `chroot` has no open directory file descriptors. Finally, UserFS ensures that a process cannot receive a directory file descriptor via file descriptor passing from outside the jail: it annotates Unix domain sockets with the sender's root directory (or a "prohibited" value if there are senders with different root directories) on `sendmsg`, and checks that the sender's root directory matches the recipient process root directory on `recvmsg`, if the message contains a directory file descriptor.

#### 3.2.2 *Firewall rules*

Ideally, we would like users to be able to run a process with a set of firewall rules attached to it, and for those firewall rules to apply to any child processes spawned by that process, much in the same way that `chroot` applies to all child processes. Unfortunately, this would require changing the core Linux kernel: at the very least, it would be necessary to track the "current firewall ruleset" for each process. Since we wanted to implement UserFS purely



| UID | Parent UID | Generation number | Owner UID | Owner GID |
|---|---|---|---|---|
| 32 bits | 32 bits | 64 bits | 32 bits | 32 bits |

**Figure 2**: Record stored by UserFS on disk for each allocated UID, totaling 24 bytes per allocated UID.

in terms of loadable kernel modules, we compromised, and associated firewall rules with UIDs instead. The kernel already keeps track of the UID for each process, and propagates the UID to the children of that process, so UserFS simply needs to ensure that firewall rules for newly-allocated UIDs inherit the firewall rules for the parent UID.

UserFS's firewall system consists of rules, which form rulesets, which are in turn associated with UIDs. At the lowest level, rules are of the form ⟨*action*, *proto*, *address*, *netmask*, *port*⟩. Our prototype supports two kinds of actions, ALLOW and BLOCK, and two protocols, TCP and UDP. The protocol, address, netmask, and port are matched against the destination of outgoing packets or the source of incoming packets; port value 0 matches any port. Supporting just TCP and UDP protocols suffices because, on Linux, a non-root process cannot open a raw socket to send arbitrary packets that are neither TCP or UDP. For kernels that support other protocols, such as SCTP, UserFS's rules could be augmented to track additional protocols.

A ruleset is an ordered sequence of rules, used to determine whether a packet should be allowed or blocked. When checking a packet against a ruleset, UserFS finds the earliest rule in the ruleset that matches the packet, and uses that rule's action to determine if the packet should be allowed or blocked. Each ruleset contains two implicit rules at the end, ⟨ALLOW, TCP, 0.0.0.0, 0.0.0.0, 0⟩ and ⟨ALLOW, UDP, 0.0.0.0, 0.0.0.0, 0⟩, which allow any packets by default. Each UID is associated with a ruleset, and applications can modify that UID's ruleset by adding or removing rules as necessary.

One potential worry in associating rulesets with a UID is that a malicious process can create a child UID with less-restrictive firewall rules. To mitigate this problem, UserFS checks not only the UID's own firewall ruleset, but also the rulesets of all parent UIDs, and only allows packets if they are allowed by every ruleset in this chain.

UserFS provides a Ufile ioctl to add or remove rules from that UID's firewall ruleset. However, there is a slight complication: on the one hand, we want to ensure that a process cannot modify its own firewall ruleset, but on the other hand, a process can always open its own Ufile. To address this problem, UserFS allows the firewall ioctl to be invoked only by the parent UID of a Ufile. This ensures that a process cannot change firewall rules for itself through its own Ufile.

## 4 Implementation

We have implemented UserFS as a kernel module for version 2.6.31 of the Linux kernel. The UserFS kernel module comprises a little less than 3,000 lines of code, excluding unit tests and the user-space `mount.userfs` command. UserFS relies heavily on the LSM framework [44] for checking generation numbers on setuid files (using file_permission and inode_setattr hooks), for confining `chroot` processes (using socket_sendmsg and socket_recvmsg hooks), and on netfilter for implementing network filtering (using `NF_INET_LOCAL_IN` and `NF_INET_LOCAL_OUT` hooks). UserFS also adds support to allow a process to `chown` or `chgrp` files between different UIDs that the process has privileges over.

Because UserFS is implemented as a kernel module, and does not modify core kernel code, it makes some trade-offs. For example, the kernel's versions of `chown`, `chgrp`, and `chroot` are not flexible enough for UserFS to implement its desired security policy from a kernel module. As a workaround, UserFS provides ioctls that implement equivalent functionality with its own security policy. Integrating UserFS into the core kernel code would both simplify our implementation and offer a more coherent interface to applications.

We have also implemented helper libraries for applications using UserFS, for both C and PHP. The C library comprises about 1,500 lines of code, including functions to execute a program in a newly-allocated jail and under a fresh user ID, to fork with a new UID, and to manipulate user IDs. The C library is careful to open all Ufiles with the `O_CLOEXEC` flag to avoid accidentally leaking Ufile file descriptors to other processes. The PHP library adds about 600 more lines on top of the C library to allow PHP applications to manipulate Ufiles.

## 5 Applying UserFS

To illustrate how UserFS would be used in practice, we modified several applications to take advantage of UserFS, including the Chromium web browser, the DokuWiki web application, Unix command-line utilities, and an FTP server. The rest of this section reports on these applications, focusing on the changes we had to make to each application in order to use UserFS, and the resulting benefits from doing so.

### 5.1 DokuWiki

Many web applications implement their own protection mechanisms, since they do not typically run as root, and thus cannot allocate user IDs for each application-level

user. This can lead to vulnerabilities if the application developers make a mistake in performing security checks [9]. To show how UserFS can prevent similar problems, we modified DokuWiki [10], a wiki application written in PHP that supports read-protected and write-protected pages [11] and that stores wiki pages in the server's file system, to enforce the protection of wiki pages using file system permissions.

Our modified version of DokuWiki allocates a separate UID for each wiki user, and sets Unix permissions on wiki page files to reflect the protection of that page (we use ACL support in the ext4 file system [19] to represent ACLs that involve multiple users). To minimize the amount of damage that an attacker can do, our modified version of DokuWiki executes each HTTP request in a separate process, and allocates a new ephemeral user ID for the initial processing of each request[2]. If an HTTP request provides the correct password for a user account, the DokuWiki PHP process handling that request can obtain a file descriptor for that user's Ufile, and change its UID to that user, by using the UserFS PHP module. This in turn allows a DokuWiki process to read or write wiki pages accessible to that user. Figure 3 shows the flow of an HTTP request in our modified DokuWiki.

One of the key parts of our modified DokuWiki is the login mechanism, which allows the DokuWiki process to obtain a file descriptor to a user's Ufile if it knows the user's password. We implemented this mechanism in a short C program called `dokusu`. `dokusu` accepts a username and password on stdin, checks the username and password against the password database, and if the password matches, it opens the corresponding user's Ufile (listed in the password database) and uses file descriptor passing to pass it back to the caller via stdout (which the caller should have set up as a Unix domain socket). `dokusu` is typically installed as a setuid program with the administrator's UID, and the permissions on all Ufiles for DokuWiki users in `/userfs` and on the password database are such that only the administrator can access them. Thus, to authenticate, DokuWiki spawns `dokusu`, passes it the username and password from the HTTP request, and waits for a Ufile in response.

DokuWiki keeps a copy of the user's password in its HTTP cookie, which makes it easy to authenticate subsequent requests. Cookies that store a session ID could also be supported, by augmenting `dokusu` to keep track of all currently valid session IDs and the corresponding user IDs for each session, and to accept a valid session ID as credentials for the corresponding user.

[2]We changed the first line of DokuWiki's PHP files to allocate a new ephemeral UID for each request, and to switch to that user ID. An alternative approach would be to modify the web server to launch each CGI script under a fresh user ID.

Making these changes to DokuWiki involved adding approximately 80 lines of PHP code, and implementing the 160-line `dokusu` program, on top of our UserFS PHP and C libraries, respectively. These changes allow the kernel to enforce DokuWiki's security policy, and Section 6.2 shows the effectiveness of this technique.

### 5.2 Command-line tools

To make it easy for ordinary users to use UserFS, we implemented a command to allocate a new user ID, called `ualloc`, which simply issues `USERFS_IOC_ALLOC` on the Ufile of the current process UID and prints the resulting UID value. To allow users to run code with these newly allocated UIDs, we modified `su` to allow users to be specified by their Ufile pathname instead of by username (in which case `su` relies on Ufile permissions to check if the caller is allowed to run as the target user, since it has no way of authenticating UserFS users by password). These modifications comprised approximately 300 lines of code.

With these changes, users can easily run arbitrary Unix applications with fewer privileges. For example, if a user wants to run a peer-to-peer file sharing program, but wants to avoid the risk of that program sharing private files with the rest of the world, the user can simply run `ualloc` to create a fresh UID for that program, run `su /userfs/`*`newuid`*`/ctl` to open a shell running as that user ID, and run the file sharing program from that shell. The file sharing program will not be able to read any of the user's private files (i.e., files that are not world-readable).

Users can also create processes that are isolated from the user's own account. For instance, `ssh-agent` stores a decrypted version of the user's SSH private key in memory. If an attacker compromises the user's account and finds a running `ssh-agent` process, the attacker can extract the key from memory by debugging `ssh-agent`. To prevent this, a user can allocate a fresh user ID with `ualloc`, run `ssh-agent` as that user ID, change permissions on the agent's socket so that the user can talk to `ssh-agent`[3], and finally change the owner of `ssh-agent`'s Ufile to `ssh-agent`'s UID, so that the user can no longer access it. The only thing the user can do at this point is to communicate with `ssh-agent` via the socket, or kill `ssh-agent` by deallocating the UID. The user cannot access `ssh-agent`'s memory to extract the key, since `ssh-agent` is running under a different UID, and the user cannot gain that UID's privileges, because it cannot open the corresponding Ufile.

Finally, UserFS makes it easier for users to switch user IDs. With traditional `su`, the user receives a new shell running under the target UID, with a new working directory, new command history, and new environment variables. When the user wants to switch back to their original UID, they again lose their command history and environment variables. To show how UserFS can help, we modified `su` to support an option to pass the resulting Ufile back to the caller via FD passing, instead of running a shell under the resulting user's UID, and likewise modified `bash` to accept the Ufile FD from `su` (much like the design of `dokusu` in the previous subsection) and invoke `USERFS_IOC_SETUID` on it. This allows the user to switch UIDs without having to switch shell processes, improving user convenience.

[3]We had to make a two-line change to `ssh-agent` to support this, since by default `ssh-agent` refuses connections from other UIDs.

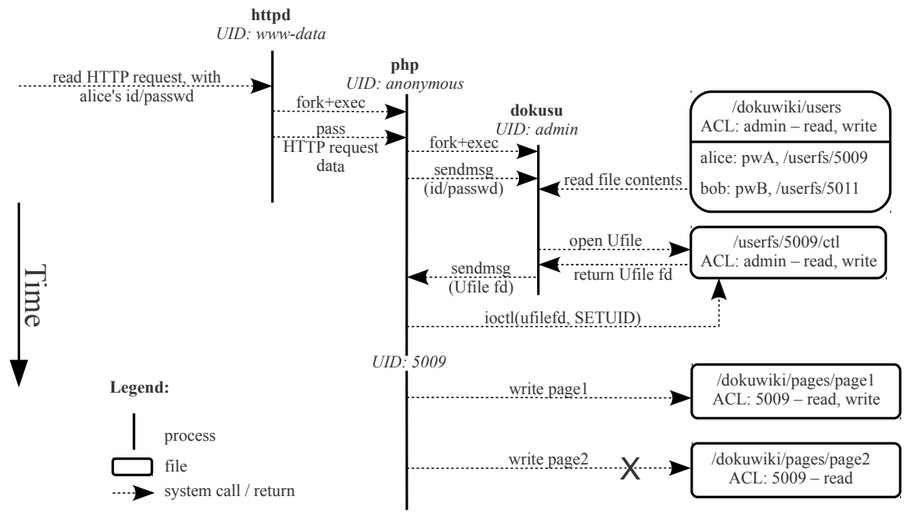



**Figure 3**: Flow of an HTTP request in our modified version of DokuWiki, showing Alice trying to write to two protected pages. Bold labels show process names (httpd, php, and dokusu). Italic labels show process UIDs (www-data, anonymous, admin, and 5009). After reading the users file, dokusu checks the supplied password against the stored password. In this example, Alice can modify page 1 (to which she has read-write access), but cannot modify page 2 (to which she has read-only access). In practice, Alice's UID would be a value between $2^{30}$ and $2^{31} - 1$, instead of 5009.

### 5.3 User authentication

Many network services run as root in order to authenticate users and to invoke `setuid` to switch to that user's UID afterwards. Unfortunately, these network services are also some of the most vulnerable components in a system, since they are directly exposed to an attacker's inputs from the network, and if they are compromised, the attacker gains root access. With UserFS, network services like ftp, ssh, telnet, or IMAP mail servers can instead run as completely unprivileged processes[4], and perform authentication and login via Unix domain sockets like in DokuWiki above. (Infact, they can reuse the `su` command from the previous subsection, which passes back the authenticated user's Ufile to the caller.) This ensures that if an attacker finds a vulnerability in a network service, they get almost no privileges on the system. To prevent an attacker from subverting subsequent connections to a compromised service, a new service process should be forked, with a fresh non-persistent UID, for each connection.

To show this is feasible, we modified the Linux NetKit FTP server [22] to authenticate users using Ufile passing; doing this required 50 lines of code, indicating that it is relatively easy to make such changes to existing applications (unlike privilege separation in the style of OpenSSH [39], which is much more invasive). Our modified FTP server uses the `su` program as its authentication agent.

### 5.4 Chromium browser

One application that is already broken up into many processes is Google's Chromium browser [2], which maintains a separate process for rendering each browser window, and a single browser kernel process responsible for coordinating with the rendering processes. This architecture easily lends itself to privilege separation, by isolating each rendering process. Indeed, Chromium already tries to do this on Windows using tokens [17], although this does not prevent a compromised browser process from accessing the network or world-accessible files.

With UserFS, browser processes can be isolated by allocating a fresh non-persistent UID for each rendering process, `chroot`ing the rendering process into an

[4]We provide setuid-root binaries to open specific TCP ports below 1024, such as port 80 for the web server, accessible only to the web server's UID.

empty directory, and setting up firewall rules that block all network traffic. Making these changes to Chromium required replacing the `fork` call in Chromium with a call to a UserFS library function called `ufork` that performs precisely the actions mentioned above[5]. All communication between the browser kernel process and the rendering processes happens via sockets, which remain intact, while the kernel's protection mechanisms ensure that a compromised rendering process cannot access any files, signal any processes, or use the network.

# 6 Evaluation

To evaluate UserFS, we first discuss its security, then show how UserFS helps prevent attackers from exploiting vulnerabilities in DokuWiki, and then measure the performance overheads associated with UserFS.

## 6.1 Kernel security

The goal of UserFS is to allow any application to use the kernel's protection mechanisms. This implicitly assumes that the kernel's mechanisms are secure. While security vulnerabilities are found in the kernel from time to time [1], this paper does not attempt to tackle this problem, and assumes that, for the time being, users will continue to run applications on the Linux kernel.

Thus, we mostly focus on the security of any changes that UserFS makes to the Linux kernel. As a first-order measure, UserFS is relatively small—less than 3,000 lines of code—which simplifies the job of auditing our code. The specific mechanisms that UserFS provides that could be misused by adversaries are the `USERFS_IOC_SETUID` ioctl, allowing a process to switch user IDs, and the `chroot` mechanism that allows non-root processes to change their root directory.

We believe the `USERFS_IOC_SETUID` mechanism is secure because it only allows a process to switch user IDs if it has an open file descriptor to the corresponding Ufile. By default, each standard user's Ufile can only be opened by that user (and by root), making it no different from the current kernel policy. Users can change permissions on Ufiles to allow other processes to open them, but again, a process can only change permissions on a Ufile that they already have access to (i.e. it was initially their UID, or it was granted to them). Applications can potentially make mistakes and leak privileges over a Ufile to another process by forgetting to close a Ufile file descriptor. The UserFS library tries to mitigate this by opening all Ufiles with the `O_CLOEXEC` flag.

The `chroot` mechanism could potentially be used recursively by an adversary to escape from a chroot jail. We believe that we have implemented sufficient safeguards against this, as described in Section 3.2.1, but we have no formal proof of their correctness.

[5] We do not provide a more fine-grained lines of code measure for the `ufork` function because it internally relies on most of the other functions provided by the UserFS library.

## 6.2 Application security

Assuming UserFS and the Linux kernel are secure, we wanted to show what security benefits applications could extract from this. To do so, we decided to check whether any previously-reported vulnerabilities for DokuWiki would have been prevented by our changes to enforce the DokuWiki security policy using file system permissions. We found several vulnerabilities for DokuWiki in the past few years that allowed an attacker to compromise DokuWiki [32–37] (as opposed to information disclosure vulnerabilities, such as printing PHP debug information, which might help an attacker in exploiting another attack vector).

Our modified version of DokuWiki (backported to an older version of DokuWiki that contained the above vulnerabilities) was able to prevent exploits of code injection [35–37], directory traversal [33], and insufficient permission check [34] vulnerabilities (5 out of 6), but did not prevent exploits of a cross-site request forgery vulnerability [32]. Although our modified version of DokuWiki contained all of the above vulnerabilities, the vulnerable code was running with limited privileges (either the web server's ephemeral per-request UID, or the UID of a specific wiki user), which prevented the attack from doing any server-side damage.

## 6.3 Performance

Performance of applications running on Linux with UserFS depends on two factors: overheads imposed by UserFS on system calls, and overheads associated with privilege-separating the application to make use of UserFS. In most cases, UserFS imposes no overheads on system calls, because the kernel executes the same exact access control checks based on UIDs with or without UserFS. One exception to this is the invocation of setuid binaries, for which UserFS checks the generation number of the setuid binary against the latest generation number for that UID. Applications that are modified to take advantage of UserFS incur two additional sources of overhead: the cost to invoke UserFS mechanisms, such as ioctls to allocate or change UIDs, and the cost of privilege-separating the application into separate Unix processes.

To evaluate these three sources of overhead, we used microbenchmarks to measure the cost of system calls affected by UserFS, and we used DokuWiki to measure the cost of privilege-separating an application with UserFS. Figure 4 shows the results of these experiments on a 2.8GHz Intel Core i7 system with 8GB RAM running a 64-bit Linux 2.6.31 kernel. As can be seen from the figure, UserFS imposes minimal overheads for both user allocation and for checking generation numbers on setuid binaries (which is dwarfed by the cost of forking a setuid program in the first place). In the case of DokuWiki, the performance overhead of privilege separation is largely dominated by the cost of spawning the `dokusu` authentication agent; we expect that having a long-running authentication agent that accepts requests over Unix domain sockets would significantly reduce the cost of running DokuWiki with UserFS. However, the costs of privilege-separation are not specific to UserFS, and have been studied before extensively [2, 3, 5–7, 24, 26, 39].

| Operation | Time without UserFS | Time with UserFS |
|---|---|---|
| Allocate UID | — | 0.022 ms |
| Check generation number of setuid executable | 0 | 0.003 ms |
| Run `sudo ls` | 10.943 ms | 10.946 ms |
| Fetch page from DokuWiki | 45 ms | 61 ms |

**Figure 4**: Time taken to perform several operations with and without UserFS.

# 7 Related work

The principle of least privilege [40] is generally recognized as a good strategy for building secure systems, and has been used by many applications in practice, including qmail [3], OpenSSH [39], OKWS [24], a number of web browsers [2, 18, 41], and others. Current Unix protection mechanisms make it difficult for non-root applications to follow the principle of least privilege, by not allowing them to create less-privileged principals. This requires developers that want less privileges to actually have more privileges by running as root, and UserFS directly addresses this problem.

It is well-known that reasoning about the safety of a computer system in the presence of setuid programs is difficult [21, 27], and there are many pitfalls in implementing safe setuid programs [4, 8]. At the lowest level, UserFS does not make it any easier to write a correct setuid program. However, we hope that UserFS makes it possible for programs that currently run as root, including setuid-root programs, to run under a less privileged UID instead, mitigating the damage from any vulnerability.

Krohn argued that applications must be given mechanisms to reduce their privileges [25], and ServiceOS [42] similarly argues for support for application-level principals in the OS kernel. Capability-based systems like KeyKOS [6, 20], and DIFC systems like Asbestos [12] and HiStar [46], allow users to create new protection domains of their own, at the cost of requiring a new OS kernel. Flume [26] shows how these ideas can be implemented on top of a Linux kernel to avoid the cost of re-implementing a new OS kernel, but Flume does not allow users to apply its protection mechanisms to unmodified existing applications. UserFS shows how the idea of egalitarian protection mechanisms can be realized in a standard Linux kernel, in a way that cleanly applies to most existing applications, and achieves many of the goals suggested by Krohn [25] and Wang [42].

The use of Ufile file descriptors to represent privileges over UIDs is inspired by capability systems [28]. Unlike traditional capability systems, which use capabilities to control access to all resources, UserFS only uses file descriptors to track the set of Ufiles currently held open by a process, and to pass Ufiles between processes. Initial access to Ufiles for opening the file descriptor, as well as access to all other resources, is controlled by Unix file permissions and other Unix mechanisms. One common problem facing capability systems is revocation of access. UserFS uses generation numbers to ensure that, once a UID has been reused, leftover file descriptors cannot gain access to that UID, since their generation numbers do not match the UID's generation number.

Although current Unix protection mechanisms are not egalitarian, many systems have used them to achieve privilege separation, at the cost of requiring some part of their system to run as root. For example, OKWS [24] shows how to build a privilege-separated web server by running a launcher as root, and Android [16] similarly uses Linux user IDs to isolate different applications on a cell phone. If these platforms start running increasingly more complex applications inside them, those applications will not have the benefit of running as root and creating their own protection domains. UserFS would address this problem.

Similarly, there have been a number of tools that help programmers privilege-separate their existing applications [5, 7, 39]. The resulting privilege-separated applications often require root privileges to actually set up protection domains, and UserFS could be used in conjunction with these tools to run privilege-separated applications without root access.

System call interposition [15] could, in principle, implement any policy that a kernel could implement. By relying on the kernel's protection mechanisms, UserFS avoids some of the pitfalls associated with system call interposition [14] and avoids runtime overhead for most operations. More importantly, UserFS illustrates what *interface* could be used by applications to allocate and manage their protection domains and set policies; the same interface could be implemented by a system call interposition system.

Bittau et al [5] propose a new kernel abstraction called an *sthread* that can execute certain pieces of an application's code in isolation from the rest of that application. The key contribution of *sthread*s was in providing a mechanism that has relatively low overhead for fine-grained

isolation of process memory, and that can be used by any processes in the system. UserFS, on the other hand, provides persistent UIDs that can be used to control access to data in the file system, and to control interactions between multiple processes in an operating system.

The Linux kernel supports several security mechanisms in addition to traditional user ID protection, such as SELinux [29] and Linux-vserver [38], but none of these mechanisms allow users to create their own protection domains and use them to protect system resources like files and devices. One protection mechanism that *is* available to users on Linux is running code in a virtual machine such as qemu. Unfortunately, this is often too coarse-grained and heavy-weight for most applications.

Taint tracking in an operating system can be used to implement certain application-level security policies; for example, SubOS [23] shows how this can be implemented on OpenBSD. Unfortunately, these mechanisms are much more invasive and impose more runtime overhead than UserFS, which simply exposes existing mechanisms in the OS kernel.

The protection mechanisms in Windows differ from those found in Unix systems. Windows protection is centered around the notion of tokens [31]. Users can create tokens that grant almost no privileges, and this is used by applications such as Chromium to sandbox untrusted code [17]. However, there is no way to create tokens with a fresh user ID (without administrative privileges to create a new user), which makes it difficult to implement controlled sharing of system resources (as opposed to complete isolation in a sandbox). Windows tokens can be passed between processes, similar to how UserFS allows passing file descriptors for Ufiles. The Windows firewall allows associating firewall rules with executables. UserFS associates firewall rules with user IDs, and inherits firewall rules on user ID creation, which ensures that a user cannot escape firewall rules by creating and running a new executable.

## 8 Limitation and Future Work

While UserFS helps applications run code with fewer privileges, it is not a panacea. Running untrusted code on a system often exposes a wider range of possibly-vulnerable interfaces than if we were simply interacting with the attacker over the network. For example, an attacker may try to exploit bugs in the kernel or in other applications running on the same machine. Nonetheless, if it is necessary to run untrusted or partially-trusted applications on a machine, UserFS helps improve security with respect to system resources.

UserFS, much like Linux itself, currently assumes that all file systems are always mounted on the same machine, and does not have a plan for translating UIDs from a file system that was originally mounted on a different machine. One possible approach to dealing with this problem may be to maintain a globally unique name of each UID (perhaps a public key), and to store on each file system a mapping table between file system UIDs and the globally unique names for those UIDs.

When a user ID is deallocated, it may be difficult to remove non-empty directories owned by that UID in the file system without root's intervention. While we have not yet implemented a solution to this problem, we imagine a system call or a setuid-root program that, upon request, recursively garbage-collects files or sub-directories owned by de-allocated UIDs from a given directory, as long as the caller has write permission on that directory.

UserFS only protects resources managed by the operating system, such as files, processes, and devices. Web applications often use databases to store their data, which UserFS cannot protect directly. In the future, we hope to explore the use of OS UIDs in a database to implement protection of data at a finer granularity (perhaps at the row level).

Our current prototype allocates user IDs, but does not separately allocate group IDs. We believe it is best to have only one kind of dynamically allocated principal, such as the 32-bit integer called the UID in UserFS. These principals can then be used to represent either users or groups, depending on the application's requirements. The GID and grouplist associated with every Unix process could then be used to represent a process that has the privileges of multiple principals at once. To support this, UserFS could provide a `USERFS_IOC_ADDGROUP` ioctl, which would add the Ufile's UID to the grouplist of the calling process. To avoid conflicts with existing groups, this ioctl should be only allowed for dynamically-allocated UIDs. In terms of file permissions, we also believe that POSIX ACLs [19] are a better alternative to the Unix user-group-other permission bits.

UserFS relies on the kernel to support 32-bit UIDs, as opposed to 16-bit UIDs from the original Unix design. Linux has supported 32-bit UIDs since kernel version 2.3.39 (January 2000), but UserFS cannot support older file systems that can only keep track of a 16-bit UID, such as the original Minix filesystem.

Our prototype faces several limitations because it is implemented as a loadable kernel module, and avoids making any extensive changes to the Linux kernel. For example, the `chroot` system call on Linux always rejects calls from non-root users, requiring UserFS to provide an alternative way of invoking `chroot`. Performing privileged operations in the kernel also requires UserFS to sometimes change the current UID of the calling process. While we believe our prototype does so safely, being able to change permission checks inside the core kernel code would be both simpler and more secure in the long term.



If UserFS was integrated into the Linux kernel, we would hope to extend our `chroot` mechanism to also allow arbitrary users to use the Linux file system namespace mechanism (a generalization of the mount table). In particular, we want to allow any process to invoke *clone* with the `CLONE_NEWNS` flag to create a new namespace, and allow a process to change its namespace using `mount --bind` if it's running as the same UID that invoked `clone(CLONE_NEWNS)`, along with restrictions on setuid binaries similar to `chroot`. Similar support could also be added to allow users to manage the system V IPC namespace (`CLONE_NEWIPC`).

Finally, if UserFS was integrated into the Linux kernel, we would also like to replace our firewall mechanism with a per-process `iptables` firewall ruleset, inherited by child processes across `fork` and `clone`. To specify new firewall rules, applications would specify a new flag to the *clone* system call to start the child process with a fresh `iptables` ruleset. To ensure that a child cannot escape from the parent's firewall rules, the child's ruleset would be chained to the parent's.

## 9 Conclusion

This paper presented UserFS, the first system to provide egalitarian OS protection mechanisms for Linux. UserFS allows any user to use existing OS protection mechanisms, including Unix user IDs, `chroot` jails, and firewalls. This both allows applications to reduce their privileges, and in many cases avoids the need for root privileges altogether.

One key idea in UserFS is representing user IDs as files in a `/proc`-like file system. This allows applications to manage user IDs much like they would any other file, without the need to introduce any new user ID management mechanisms. UserFS maintains a hierarchy of user IDs for accountability and resource revocation purposes, but allows child user IDs in the hierarchy to be made inaccessible to parent user IDs, in order to protect sensitive processes like `ssh-agent` from outside interference. To cope with a limited 32-bit user ID namespace, UserFS introduces per-UID generation numbers that disambiguate multiple instances of a reused 32-bit UID value. Finally, UserFS implements security checks that make it safe to allow non-root users to invoke `chroot`, without allowing users to escape out of existing `chroot` jails or abuse setuid executables.

An important goal of the UserFS design is compatibility with existing applications, interfaces, and kernel components. Porting applications to use UserFS requires only tens to hundreds of lines of code, and prevents attackers from exploiting application-level vulnerabilities, such as code injection or missing ACL checks in a PHP-based wiki web application. UserFS requires minimal changes to the Linux kernel, comprising of a single 3,000-line kernel module, and incurs no performance overhead for most operations.

## Acknowledgments

We thank the anonymous reviewers, Ramesh Chandra, Chris Laas, and Xi Wang for providing valuable feedback that improved this paper. This work was supported in part by Quanta Computer. Taesoo Kim is partially supported by the Samsung Scholarship Foundation.

## References

[1] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, March 2009.

[2] Adam Barth, Collin Jackson, Charles Reis, and Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, Google Inc., 2008.

[3] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, November 2007.

[4] Matt Bishop. How to write a setuid program. *;login: The Magazine of Usenix & Sage*, 12(1):5–11, January/February 1987.

[5] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation*, pages 309–322, San Francisco, CA, April 2008.

[6] Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, April 1992.

[7] David Brumley and Dawn Xiaodong Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th Usenix Security Symposium*, pages 57–72, San Diego, CA, August 2004.

[8] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th Usenix Security Symposium*, San Francisco, CA, August 2002.

[9] Michael Dalton, Nickolai Zeldovich, and Christos Kozyrakis. Nemesis: Preventing authentication and access control vulnerabilities in web applications. In *Proceedings of the 18th Usenix Security Symposium*, pages 267–282, Montreal, Canada, August 2009.

[10] DokuWiki. `http://www.dokuwiki.org/dokuwiki`.

[11] DokuWiki. Access control lists. `http://www.dokuwiki.org/acl`.

[12] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, M. Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30, Brighton, UK, October 2005.

[13] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[14] Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003.

[15] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2004.

[16] Google, Inc. Android: Security and permissions. `http://developer.android.com/guide/topics/security/security.html`.

[17] Google, Inc. Chromium sandbox. `http://dev.chromium.org/developers/design-documents/sandbox`.

[18] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 402–416, Oakland, CA, 2008.

[19] Andreas Grünbacher. POSIX access control lists on Linux. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX track*, pages 259–272, San Antonio, TX, June 2003.

[20] Norman Hardy. KeyKOS architecture. *ACM SIGOPS Operating System Review*, 19(4):8–25, October 1985.

[21] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[22] David A. Holland. linux-ftpd. In *Linux NetKit*. `ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/linux-ftpd-0.17.tar.gz`.

[23] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan Smith. Sub-operating systems: A new approach to application security. In *SIGOPS European Workshop*, September 2002.

[24] Maxwell Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, June–July 2004.

[25] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, M. Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005.

[26] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 321–334, Stevenson, WA, October 2007.

[27] Tim Levin, Steven J. Padilla, and Cynthia E. Irvine. A formal model for UNIX setuid. In *Proceedings of the 10th IEEE Symposium on Security and Privacy*, pages 73–83, Oakland, CA, May 1989.

[28] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[29] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 29–40, June 2001. FREENIX track.

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[31] Microsoft Corp. Access tokens (windows). `http://msdn.microsoft.com/en-us/library/aa374909%28VS.85%29.aspx`.

[32] MITRE Corporation. DokuWiki cross-site request forgery vulnerability. In *Common Vulnerabilities and Exposures (CVE) database*. CVE-2010-0289.

[33] MITRE Corporation. DokuWiki directory traversal vulnerability. In *Common Vulnerabilities and Exposures (CVE) database*. CVE-2010-0287.

[34] MITRE Corporation. DokuWiki insufficient permission checking vulnerability. In *Common Vulnerabilities and Exposures (CVE) database*. CVE-2010-0288.

[35] MITRE Corporation. DokuWiki php code inclusion vulnerability. In *Common Vulnerabilities and Exposures (CVE) database*. CVE-2009-1960.

[36] MITRE Corporation. DokuWiki php code injection vulnerability. In *Common Vulnerabilities and Exposures (CVE) database*. CVE-2006-4674.

[37] MITRE Corporation. DokuWiki php code upload vulnerability. In *Common Vulnerabilities and Exposures (CVE) database*. CVE-2006-4675.

[38] Herbert Pötzl. *Linux-VServer Technology*, 2004. `http://linux-vserver.org/Linux-VServer-Paper`.

[39] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th Usenix Security Symposium*, Washington, DC, August 2003.

[40] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[41] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the Gazelle web browser. In *18th USENIX Security Symposium*, August 2009.

[42] Helen J. Wang, Alexander Moshchuk, and Alan Bush. Convergence of desktop and web applications on a multi-service OS. In *4th Usenix Workshop on Hot Topics in Security*, August 2009.

[43] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the 1st USENIX Workshop on Offensive Technologies*, Boston, MA, August 2007.

[44] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *Proceedings of the 11th Usenix Security Symposium*, San Francisco, CA, August 2002.

[45] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[46] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

# Capsicum: practical capabilities for UNIX

Robert N. M. Watson
*University of Cambridge*

Jonathan Anderson
*University of Cambridge*

Ben Laurie
*Google UK Ltd.*

Kris Kennaway
*Google UK Ltd.*

## Abstract

Capsicum is a lightweight operating system capability and sandbox framework planned for inclusion in FreeBSD 9. Capsicum extends, rather than replaces, UNIX APIs, providing new kernel primitives (sandboxed *capability mode* and *capabilities*) and a userspace sandbox API. These tools support compartmentalisation of monolithic UNIX applications into logical applications, an increasingly common goal supported poorly by discretionary and mandatory access control. We demonstrate our approach by adapting core FreeBSD utilities and Google's Chromium web browser to use Capsicum primitives, and compare the complexity and robustness of Capsicum with other sandboxing techniques.

## 1 Introduction

Capsicum is an API that brings capabilities to UNIX. Capabilities are unforgeable tokens of authority, and have long been the province of research operating systems such as PSOS [16] and EROS [23]. UNIX systems have less fine-grained access control than capability systems, but are very widely deployed. By adding capability primitives to standard UNIX APIs, Capsicum gives application authors a realistic adoption path for one of the ideals of OS security: least-privilege operation. We validate our approach through an open source prototype of Capsicum built on (and now planned for inclusion in) FreeBSD 9.

Today, many popular security-critical applications have been decomposed into parts with different privilege requirements, in order to limit the impact of a single vulnerability by exposing only limited privileges to more risky code. Privilege separation [17], or *compartmentalisation*, is a pattern that has been adopted for applications such as OpenSSH, Apple's SecurityServer, and, more recently, Google's Chromium web browser. Compartmentalisation is enforced using various access control techniques, but only with significant programmer effort and significant technical limitations: current OS facilities are simply not designed for this purpose.

The access control systems in conventional (non-capability-oriented) operating systems are *Discretionary Access Control* (DAC) and *Mandatory Access Control* (MAC). DAC was designed to protect users from each other: the owner of an object (such as a file) can specify *permissions* for it, which are checked by the OS when the object is accessed. MAC was designed to enforce system policies: system administrators specify policies (e.g. "users cleared to Secret may not read Top Secret documents"), which are checked via run-time hooks inserted into many places in the operating system's kernel.

Neither of these systems was designed to address the case of a single application processing many types of information on behalf of one user. For instance, a modern web browser must parse HTML, scripting languages, images and video from many untrusted sources, but because it acts with the full power of the user, has access to all his or her resources (such implicit access is known as ambient authority).

In order to protect user data from malicious JavaScript, Flash, etc., the Chromium web browser is decomposed into several OS processes. Some of these processes handle content from untrusted sources, but their access to user data is restricted using DAC or MAC mechanism (the process is *sandboxed*).

These mechanisms vary by platform, but all require a significant amount of programmer effort (from hundreds of lines of code or policy to, in one case, 22,000 lines of C++) and, sometimes, elevated privilege to bootstrap them. Our analysis shows significant vulnerabilities in all of these sandbox models due to inherent flaws or incorrect use (see Section 5).

Capsicum addresses these problems by introducing new (and complementary) security primitives to support compartmentalisation: *capability mode* and *capabilities*. Capsicum capabilities should not be confused with operating system privileges, occasionally referred to as ca-

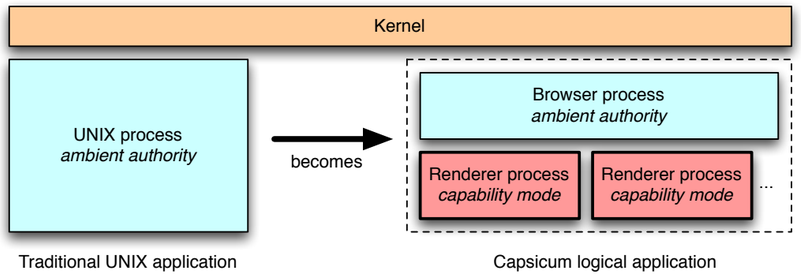



Figure 1: Capsicum helps applications self-compartmentalise.

pabilities in the OS literature. Capsicum capabilities are an extension of UNIX file descriptors, and reflect rights on specific objects, such as files or sockets. Capabilities may be delegated from process to process in a granular way in the same manner as other file descriptor types: via inheritance or message-passing. Operating system privilege, on the other hand, refers to exemption from access control or integrity properties granted to processes (perhaps assigned via a role system), such as the right to override DAC permissions or load kernel modules. A fine-grained privilege policy supplements, but does not replace, a capability system such as Capsicum. Likewise, DAC and MAC can be valuable components of a system security policy, but are inadequate in addressing the goal of application privilege separation.

We have modified several applications, including base FreeBSD utilities and Chromium, to use Capsicum primitives. No special privilege is required, and code changes are minimal: the `tcpdump` utility, plagued with security vulnerabilities in the past, can be sandboxed with Capsicum in around ten lines of code, and Chromium can have OS-supported sandboxing in just 100 lines.

In addition to being more secure and easier to use than other sandboxing techniques, Capsicum performs well: unlike pure capability systems where system calls necessarily employ message passing, Capsicum's capability-aware system calls are just a few percent slower than their UNIX counterparts, and the `gzip` utility incurs a constant-time penalty of 2.4 ms for the security of a Capsicum sandbox (see Section 6).

## 2 Capsicum design

Capsicum is designed to blend capabilities with UNIX. This approach achieves many of the benefits of least-privilege operation, while preserving existing UNIX APIs and performance, and presents application authors with an adoption path for capability-oriented design.

Capsicum extends, rather than replaces, standard UNIX APIs by adding kernel-level primitives (a sandboxed *capability mode*, *capabilities* and others) and userspace support code (*libcapsicum* and a *capability-aware run-time linker*). Together, these extensions support application *compartmentalisation*, the decomposition of monolithic application code into components that will run in independent sandboxes to form *logical applications*, as shown in Figure 1.

Capsicum requires application modification to exploit new security functionality, but this may be done gradually, rather than requiring a wholesale conversion to a pure capability model. Developers can select the changes that maximise positive security impact while minimising unacceptable performance costs; where Capsicum replaces existing sandbox technology, a performance improvement may even be seen.

This model requires a number of pragmatic design choices, not least the decision to eschew micro-kernel architecture and migration to pure message-passing. While applications may adopt a message-passing approach, and indeed will need to do so to fully utilise the Capsicum architecture, we provide "fast paths" in the form of direct system call manipulation of kernel objects through delegated file descriptors. This allows native UNIX performance for file system I/O, network access, and other critical operations, while leaving the door open to techniques such as message-passing system calls for cases where that proves desirable.

### 2.1 Capability mode

Capability mode is a process credential flag set by a new system call, `cap_enter`; once set, the flag is inherited by all descendent processes, and cannot be cleared. Processes in capability mode are denied access to global namespaces such as the filesystem and PID namespaces (see Figure 2). In addition to these namespaces, there are several system management interfaces that must be protected to maintain UNIX process isolation. These interfaces include `/dev` device nodes that allow physical memory or PCI bus access, some `ioctl` operations on sockets, and management interfaces such as `reboot` and `kldload`, which loads kernel modules.

Access to system calls in capability mode is also restricted: some system calls requiring global namespace access are unavailable, while others are constrained. For instance, `sysctl` can be used to query process-local information such as address space layout, but also to monitor a system's network connections. We have constrained `sysctl` by explicitly marking ≈30 of 3000 parameters as permitted in capability mode; all others are denied.

The system calls which require constraints are `sysctl`, `shm_open`, which is permitted to create *anonymous memory objects*, but not named ones, and the `openat` family of system calls. These calls already accept a file descriptor argument as the directory to perform the open, rename, etc. relative to; in capability mode, they are constrained so that they can only operate on objects "under" this descriptor. For instance, if file descriptor 4 is a capability allowing access to /lib, then `openat(4, "libc.so.7")` will succeed, whereas `openat(4, "../etc/passwd")` and `openat(4, "/etc/passwd")` will not.

### 2.2 Capabilities

The most critical choice in adding capability support to a UNIX system is the relationship between capabilities and file descriptors. Some systems, such as Mach/BSD, have maintained entirely independent notions: Mac OS X provides each task with both indexed capabilities (ports) and file descriptors. Separating these concerns is logical, as Mach ports have different semantics from file descriptors; however, confusing results can arise for application developers dealing with both Mach and BSD APIs, and we wanted to reuse existing APIs as much as possible. As a result, we chose to extend the file descriptor abstraction, and introduce a new file descriptor type, the capability, to wrap and protect raw file descriptors.

File descriptors already have some properties of capabilities: they are unforgeable tokens of authority, and can be inherited by a child process or passed between processes that share an IPC channel. Unlike "pure" capabilities, however, they confer very broad rights: even if a file descriptor is read-only, operations on meta-data such as `fchmod` are permitted. In the Capsicum model, we restrict these operations by wrapping the descriptor in a capability and permitting only authorised operations via the capability, as shown in Figure 3.

The `cap_new` system call creates a new capability given an existing file descriptor and a mask of rights; if the original descriptor is a capability, the requested rights must be a subset of the original rights. Capability rights are checked by `fget`, the in-kernel code for converting file descriptor arguments to system calls into in-kernel references, giving us confidence that no paths exist to access file descriptors without capability checks. Capability file descriptors, as with most others in the system, may be inherited across `fork` and `exec`, as well as passed via UNIX domain sockets.

There are roughly 60 possible mask rights on each capability, striking a balance between message-passing (two rights: send and receive), and MAC systems (hundreds of access control checks). We selected rights to align with logical methods on file descriptors: system calls implementing semantically identical operations require the same rights, and some calls may require multiple rights. For example, `pread` (read to memory) and `preadv` (read to a memory vector) both require `CAP_READ` in a capability's rights mask, and `read` (read bytes using the file offset) requires `CAP_READ | CAP_SEEK` in a capability's rights mask.

Capabilities can wrap any type of file descriptor including directories, which can then be passed as arguments to `openat` and related system calls. The `*at` system calls begin relative lookups for file operations with the directory descriptor; we disallow some cases when a capability is passed: absolute paths, paths containing ".." components, and `AT_FDCWD`, which requests a lookup relative to the current working directory. With these constraints, directory capabilities delegate file system namespace subsets, as shown in Figure 4. This allows sandboxed processes to access multiple files in a directory (such as the library path) without the performance overhead or complexity of proxying each file `open` via IPC to a process with ambient authority.

The ".." restriction is a conservative design, and prevents a subtle problem similar to historic `chroot` vulnerabilities. A single directory capability that only enforces containment by preventing ".." lookup on the root of a subtree operates correctly; however, two colluding sandboxes (or a single sandbox with two capabilities) can race to actively rearrange a tree so that the check always succeeds, allowing escape from a delegated subset. It is possible to imagine less conservative solutions, such as preventing upward renames that could introduce exploitable cycles during lookup, or additional synchronisation; these strike us as more risky tactics, and we have selected the simplest solution, at some cost to flexibility.

Many past security extensions have composed poorly with UNIX security leading to vulnerabilities; thus, we disallow privilege elevation via `fexecve` using setuid and setgid binaries in capability mode. This restriction does not prevent setuid binaries from using sandboxes.

| Namespace | Description |
|---|---|
| Process ID (PID) | UNIX processes are identified by unique IDs. PIDs are returned by `fork` and used for signal delivery, debugging, monitoring, and status collection. |
| File paths | UNIX files exist in a global, hierarchical namespace, which is protected by discretionary and mandatory access control. |
| NFS file handles | The NFS client and server identify files and directories on the wire using a flat, global file handle namespace. They are also exposed to processes to support the lock manager daemon and optimise local file access. |
| File system ID | File system IDs supplement paths to mount points, and are used for forceable unmount when there is no valid path to the mount point. |
| Protocol addresses | Protocol families use socket addresses to name local and foreign endpoints. These exist in global namespaces, such as IPv4 addresses and ports, or the file system namespace for local domain sockets. |
| Sysctl MIB | The `sysctl` management interface uses numbered and named entries, used to get or set system information, such as process lists and tuning parameters. |
| System V IPC | System V IPC message queues, semaphores, and shared memory segments exist in a flat, global integer namespace. |
| POSIX IPC | POSIX defines similar semaphore, message queue, and shared memory APIs, with an undefined namespace: on some systems, these are mapped into the file system; on others they are simply a flat global namespaces. |
| System clocks | UNIX systems provide multiple interfaces for querying and manipulating one or more system clocks or timers. |
| Jails | The management namespace for FreeBSD-based virtualised environments. |
| CPU sets | A global namespace for affinity policies assigned to processes and threads. |

Figure 2: Global namespaces in the FreeBSD operating kernel

### 2.3 Run-time environment

Even with Capsicum's kernel primitives, creating sandboxes without leaking undesired resources via file descriptors, memory mappings, or memory contents is difficult. `libcapsicum` therefore provides an API for starting scrubbed sandbox processes, and explicit delegation APIs to assign rights to sandboxes. `libcapsicum` cuts off the sandbox's access to global namespaces via `cap_enter`, but also closes file descriptors not positively identified for delegation, and flushes the address space via `fexecve`. Sandbox creation returns a UNIX domain socket that applications can use for inter-process communication (IPC) between host and sandbox; it can also be used to grant additional rights as the sandbox runs.

## 3 Capsicum implementation

### 3.1 Kernel changes

Many system call and capability constraints are applied at the point of implementation of kernel services, rather than by simply filtering system calls. The advantage of this approach is that a single constraint, such as the blocking of access to the global file system namespace, can be implemented in one place, `namei`, which is responsible for processing all path lookups. For example, one might not have expected the `fexecve` call to cause global namespace access, since it takes a file descriptor as its argument rather than a path for the binary to execute. However, the file passed by file descriptor specifies its run-time linker via a path embedded in the binary, which the kernel will then open and execute.

Similarly, capability rights are checked by the kernel function `fget`, which converts a numeric descriptor into a `struct file` reference. We have added a new `rights` argument, allowing callers to declare what capability rights are required to perform the current operation. If the file descriptor is a raw UNIX descriptor, or wrapped by a capability with sufficient rights, the operation succeeds. Otherwise, `ENOTCAPABLE` is returned. Changing the signature of `fget` allows us to use the compiler to detect missed code paths, providing greater assurance that all cases have been handled.

One less trivial global namespace to handle is the process ID (PID) namespace, which is used for process creation, signalling, debugging and exit status, critical operations for a logical application. Another problem for logical applications is that libraries cannot create and manage worker processes without interfering with process management in the application itself—unexpected



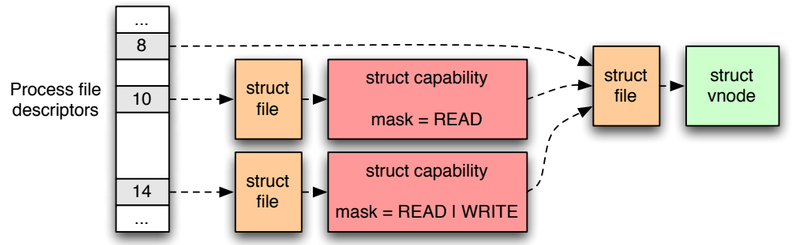



Figure 3: Capabilities "wrap" normal file descriptors, masking the set of permitted methods.

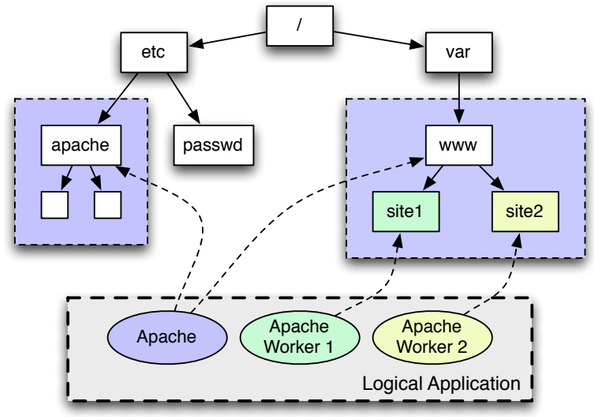



Figure 4: Portions of the global filesystem namespace can be delegated to sandboxed processes.

`SIGCHLD` signals are delivered to the application, and unexpected process IDs are returned by `wait`.

Process descriptors address these problems in a manner similar to Mach task ports: creating a process with `pdfork` returns a file descriptor to use for process management tasks, such as monitoring for exit via `poll`. When the process descriptor is closed, the process is terminated, providing a user experience consistent with that of monolithic processes: when a user hits Ctrl-C, or the application segfaults, all processes in the logical application terminate. Termination does not occur if reference cycles exist among processes, suggesting the need for a new "logical application" primitive—see Section 7.

### 3.2 The Capsicum run-time environment

Removing access to global namespaces forces fundamental changes to the UNIX run-time environment. Even the most basic UNIX operations for starting processes and running programs have been eliminated: `fork` and `exec` both rely on global namespaces. Responsibility for launching a sandbox is shared. `libcapsicum` is invoked by the application, and responsible for forking a new process, gathering together delegated capabilities from both the application and run-time linker, and directly executing the run-time linker, passing the sandbox binary via a capability. ELF headers normally contain a hard-coded path to the run-time linker to be used with the binary. We execute the Capsicum-aware run-time linker directly, eliminating this dependency on the global file system namespace.

Once `rtld-elf-cap` is executing in the new process, it loads and links the binary using libraries loaded via library directory capabilities set up by `libcapsicum`. The `main` function of a program can call `lcs_get` to determine whether it is in a sandbox, retrieve sandbox state,

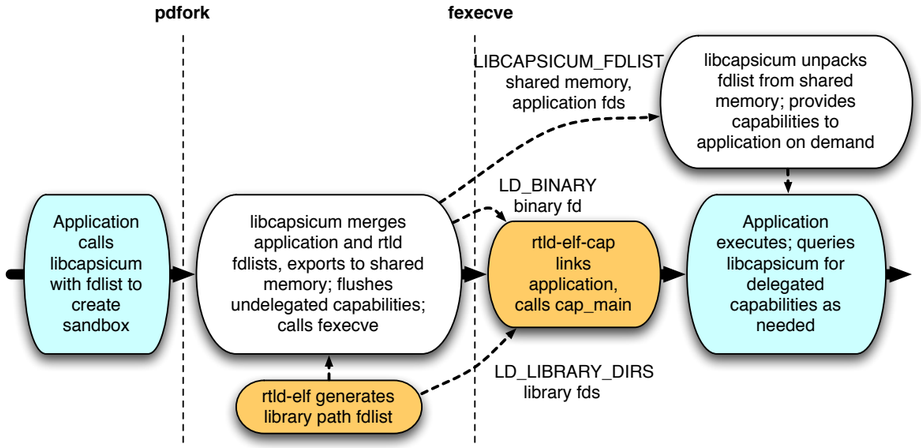



Figure 5: Process and components involved in creating a new `libcapsicum` sandbox

query creation-time delegated capabilities, and retrieve an IPC handle so that it can process RPCs and receive run-time delegated capabilities. This allows a single binary to execute both inside and outside of a sandbox, diverging behaviour based on its execution environment. This process is illustrated in greater detail in Figure 5.

Once in execution, the application is linked against normal C libraries and has access to much of the traditional C run-time, subject to the availability of system calls that the run-time depends on. An IPC channel, in the form of a UNIX domain socket, is set up automatically by `libcapsicum` to carry RPCs and capabilities delegated after the sandbox starts. Capsicum does not provide or enforce the use of a specific Interface Description Language (IDL), as existing compartmentalised or privilege-separated applications have their own, often hand-coded, RPC marshalling already. Here, our design choice differs from historic capability systems, which universally have selected a specific IDL, such as the Mach Interface Generator (MIG) on Mach.

`libcapsicum`'s `fdlist` (file descriptor list) abstraction allows complex, layered applications to declare capabilities to be passed into sandboxes, in effect providing a sandbox template mechanism. This avoids encoding specific file descriptor numbers into the ABI between applications and their sandbox components, a technique used in Chromium that we felt was likely to lead to programming errors. Of particular concern is hard-coding of file descriptor numbers for specific purposes, when those descriptor numbers may already have been used by other layers of the system. Instead, application and library components declare process-local names bound to file descriptor numbers before creating the sandbox; matching components in the sandbox can then query those names to retrieve (possibly renumbered) file descriptors.

## 4 Adapting applications to use Capsicum

Adapting applications for use with sandboxing is a non-trivial task, regardless of the framework, as it requires analysing programs to determine their resource dependencies, and adopting a distributed system programming style in which components must use message passing or explicit shared memory rather than relying on a common address space for communication. In Capsicum, programmers have a choice of working directly with capability mode or using `libcapsicum` to create and manage sandboxes, and each model has its merits and costs in terms of development complexity, performance impact, and security:

1. Modify applications to use `cap_enter` directly in order to convert an existing process with ambient privilege into a capability mode process inheriting only specific capabilities via file descriptors and virtual memory mappings. This works well for applications with a simple structure like: open all resources, then process them in an I/O loop, such as programs operating in a UNIX pipeline, or interacting with the network for the purposes of a single connection. The performance overhead will typically be extremely low, as changes consist of encapsulating broad file descriptor rights into capabilities, followed by entering capability mode. We illustrate this approach with `tcpdump`.

2. Use `cap_enter` to reinforce the sandboxes of applications with existing privilege separation or compartmentalisation. These applications have a more complex structure, but are already aware that some access limitations are in place, so have already been designed with file descriptor passing in mind. Refining these sandboxes can significantly improve security in the event of a vulnerability, as we show for `dhclient` and Chromium; the performance and complexity impact of these changes will be low because the application already adopts a message passing approach.

3. Modify the application to use the full `libcapsicum` API, introducing new compartmentalisation or reformulating existing privilege separation. This offers significantly stronger protection, by virtue of flushing capability lists and residual memory from the host environment, but at higher development and run-time costs. Boundaries must be identified in the application such that not only is security improved (i.e., code processing risky data is isolated), but so that resulting performance is sufficiently efficient. We illustrate this technique using modifications to `gzip`.

Compartmentalised application development is, of necessity, distributed application development, with software components running in different processes and communicating via message passing. Distributed debugging is an active area of research, but commodity tools are unsatisfying and difficult to use. While we have not attempted to extend debuggers, such as `gdb`, to better support distributed debugging, we have modified a number of FreeBSD tools to improve support for Capsicum development, and take some comfort in the generally synchronous nature of compartmentalised applications.

The FreeBSD `procstat` command inspects kernel-related state of running processes, including file descriptors, virtual memory mappings, and security credentials. In Capsicum, these resource lists become capability lists, representing the rights available to the process. We have extended `procstat` to show new Capsicum-related information, such as capability rights masks on file descriptors and a flag in process credential listings to indicate capability mode. As a result, developers can directly inspect the capabilities inherited or passed to sandboxes.

When adapting existing software to run in capability mode, identifying capability requirements can be tricky; often the best technique is to discover them through dynamic analysis, identifying missing dependencies by tracing real-world use. To this end, capability-related failures return a new `errno` value, `ENOTCAPABLE`, distinguishing them from other failures, and system calls such as `open` are blocked in `namei`, rather than the system call boundary, so that paths are shown in FreeBSD's `ktrace` facility, and can be utilised in `DTrace` scripts.

Another common compartmentalised development strategy is to allow the multi-process logical application to be run as a single process for debugging purposes. `libcapsicum` provides an API to query whether sandboxing for the current application or component is enabled by policy, making it easy to enable and disable sandboxing for testing. As RPCs are generally synchronous, the thread stack in the sandbox process is logically an extension of the thread stack in the host process, which makes the distributed debugging task less fraught than it otherwise might appear.

### 4.1 tcpdump

`tcpdump` provides an excellent example of Capsicum primitives offering immediate wins through straightforward changes, but also the subtleties that arise when compartmentalising software not written with that goal in mind. `tcpdump` has a simple model: compile a pattern into a BPF filter, configure a BPF device as an input source, and loop writing captured packets rendered as text. This structure lends itself to sandboxing: resources are acquired early with ambient privilege, and later processing depends only on held capabilities, so can execute in capability mode. The two-line change shown in Figure 6 implements this conversion.

This significantly improves security, as historically fragile packet-parsing code now executes with reduced privilege. However, further analysis with the `procstat` tool is required to confirm that only desired capabilities are exposed. While there are few surprises, unconstrained access to a user's terminal connotes significant rights, such as access to key presses. A refinement, shown in Figure 7, prevents reading `stdin` while still allowing output. Figure 8 illustrates `procstat` on the resulting process, including capabilities wrapping file descriptors in order to narrow delegated rights.

`ktrace` reveals another problem, `libc` DNS resolver code depends on file system access, but not until after `cap_enter`, leading to denied access and lost functionality, as shown in Figure 9.

This illustrates a subtle problem with sandboxing: highly layered software designs often rely on on-demand initialisation, lowering or avoiding startup costs, and those initialisation points are scattered across many components in system and application code. This is corrected by switching to the lightweight resolver, which sends DNS queries to a local daemon that performs actual res-

```
+       if (cap_enter() < 0)
+               error("cap_enter: %s", pcap_strerror(errno));
        status = pcap_loop(pd, cnt, callback, pcap_userdata);
```

Figure 6: A two-line change adding capability mode to `tcpdump`: `cap_enter` is called prior to the main `libpcap` (packet capture) work loop. Access to global file system, IPC, and network namespaces is restricted.

```
+       if (lc_limitfd(STDIN_FILENO, CAP_FSTAT) < 0)
+               error("lc_limitfd: unable to limit STDIN_FILENO");
+       if (lc_limitfd(STDOUT_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
+               error("lc_limitfd: unable to limit STDOUT_FILENO");
+       if (lc_limitfd(STDERR_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
+               error("lc_limitfd: unable to limit STDERR_FILENO");
```

Figure 7: Using `lc_limitfd`, `tcpdump` can further narrow rights delegated by inherited file descriptors, such as limiting permitted operations on `STDIN` to `fstat`.

```
  PID COMM               FD T     FLAGS CAPABILITIES PRO NAME
 1268 tcpdump             0 v rw------c           fs -   /dev/pts/0
 1268 tcpdump             1 v -w------c     wr,se,fs -   /dev/null
 1268 tcpdump             2 v -w------c     wr,se,fs -   /dev/null
 1268 tcpdump             3 v rw-------           - -   /dev/bpf
```

Figure 8: `procstat -fC` displays capabilities held by a process; `FLAGS` represents the file open flags, whereas `CAPABILITIES` represents the capabilities rights mask. In the case of `STDIN`, only `fstat` (`fs`) has been granted.

```
 1272 tcpdump CALL  open(0x80092477c,O_RDONLY,<unused>0x1b6)
 1272 tcpdump NAMI  "/etc/resolv.conf"
 1272 tcpdump RET   connect -1 errno 78 Function not implemented
 1272 tcpdump CALL  socket(PF_INET,SOCK_DGRAM,IPPROTO_UDP)
 1272 tcpdump RET   socket 4
 1272 tcpdump CALL  connect(0x4,0x7fffffffe080,0x10)
 1272 tcpdump RET   connect -1 errno 78 Function not implemented
```

Figure 9: `ktrace` reveals a problem: DNS resolution depends on file system and TCP/IP namespaces after `cap_enter`.

```
  PID COMM               FD T     FLAGS CAPABILITIES PRO NAME
18988 dhclient           0 v rw-------           - -   /dev/null
18988 dhclient           1 v rw-------           - -   /dev/null
18988 dhclient           2 v rw-------           - -   /dev/null
18988 dhclient           3 s rw-------           - UDD /var/run/logpriv
18988 dhclient           5 s rw-------           - ?
18988 dhclient           6 p rw-------           - -   -
18988 dhclient           7 v -w-------           - -   /var/db/dhclient.leas
18988 dhclient           8 v rw-------           - -   /dev/bpf
18988 dhclient           9 s rw-------           - IP? 0.0.0.0:0 0.0.0.0:0
```

Figure 10: Capabilities held by `dhclient` before Capsicum changes: several unnecessary rights are present.



olution, addressing both file system and network address namespace concerns. Despite these limitations, this example of capability mode and capability APIs shows that even minor code changes can lead to dramatic security improvements, especially for a critical application with a long history of security problems.

## 4.2 dhclient

FreeBSD ships the OpenBSD DHCP client, which includes privilege separation support. On BSD systems, the DHCP client must run with privilege to open BPF descriptors, create raw sockets, and configure network interfaces. This creates an appealing target for attackers: network code exposed to a complex packet format while running with root privilege. The DHCP client is afforded only weak tools to constrain operation: it starts as the root user, opens the resources its unprivileged component will require (raw socket, BPF descriptor, lease configuration file), forks a process to continue privileged activities (such as network configuration), and then confines the parent process using `chroot` and the `setuid` family of system calls. Despite hardening of the BPF `ioctl` interface to prevent reattachment to another interface or reprogramming the filter, this confinement is weak; `chroot` limits only file system access, and switching credentials offers poor protection against weak or incorrectly configured DAC protections on the `sysctl` and PID namespaces.

Through a similar two-line change to that in `tcpdump`, we can reinforce (or, through a larger change, replace) existing sandboxing with capability mode. This instantly denies access to the previously exposed global namespaces, while permitting continued use of held file descriptors. As there has been no explicit flush of address space, memory, or file descriptors, it is important to analyze what capabilities have been leaked into the sandbox, the key limitation to this approach. Figure 10 shows a `procstat -fC` analysis of the file descriptor array.

The existing `dhclient` code has done an effective job at eliminating directory access, but continues to allow the sandbox direct rights to submit arbitrary log messages to `syslogd`, modify the lease database, and a raw socket on which a broad variety of operations could be performed. The last of these is of particular interest due to `ioctl`; although `dhclient` has given up system privilege, many network socket `ioctl`s are defined, allowing access to system information. These are blocked in Capsicum's capability mode.

It is easy to imagine extending existing privilege separation in `dhclient` to use the Capsicum capability facility to further constrain file descriptors inherited in the sandbox environment, for example, by limiting use of the IP raw socket to `send` and `recv`, disallowing `ioctl`. Use of the `libcapsicum` API would require more significant code changes, but as `dhclient` already adopts a message passing structure to communicate with its components, it would be relatively straight forward, offering better protection against capability and memory leakage. Further migration to message passing would prevent arbitrary log messages or direct unformatted writes to `dhclient.leases.em` by constraining syntax.

## 4.3 gzip

The `gzip` command line tool presents an interesting target for conversion for several reasons: it implements risky compression/decompression routines that have suffered past vulnerabilities, it contains no existing compartmentalisation, and it executes with ambient user (rather than system) privileges. Historic UNIX sandboxing techniques, such as `chroot` and ephemeral UIDs are a poor match because of their privilege requirement, but also because (unlike with dhclient), there's no expectation that a single sandbox exist—many `gzip` sessions can run independently for many different users, and there can be no assumption that placing them in the same sandbox provides the desired security properties.

The first step is to identify natural fault lines in the application: for example, code that requires ambient privilege (due to opening files or building network connections), and code that performs more risky activities, such as parsing data and managing buffers. In `gzip`, this split is immediately obvious: the main run loop of the application processes command line arguments, identifies streams and objects to perform processing on and send results to, and then feeds them to compress routines that accept input and output file descriptors. This suggests a partitioning in which pairs of descriptors are submitted to a sandbox for processing after the ambient privilege process opens them and performs initial header handling.

We modified `gzip` to use `libcapsicum`, intercepting three core functions and optionally proxying them using RPCs to a sandbox based on policy queried from `libcapsicum`, as shown in Figure 11. Each RPC passes two capabilities, for input and output, to the sandbox, as well as miscellaneous fields such as returned size, original filename, and modification time. By limiting capability rights to a combination of `CAP_READ`, `CAP_WRITE`, and `CAP_SEEK`, a tightly constrained sandbox is created, preventing access to any other files in the file system, or other globally named resources, in the event a vulnerability in compression code is exploited.

These changes add 409 lines (about 16%) to the size of the `gzip` source code, largely to marshal and un-marshal RPCs. In adapting `gzip`, we were initially surprised to see a performance improvement; investigation of this unlikely result revealed that we had failed to propagate the

| Function | RPC | Description |
|---|---|---|
| gz_compress | PROXIED_GZ_COMPRESS | zlib-based compression |
| gz_uncompress | PROXIED_GZ_UNCOMPRESS | zlib-based decompression |
| unbzip2 | PROXIED_UNBZIP2 | bzip2-based decompression |

Figure 11: Three `gzip` functions are proxied via RPC to the sandbox

compression level (a global variable) into the sandbox, leading to the incorrect algorithm selection. This serves as reminder that code not originally written for decomposition requires careful analysis. Oversights such as this one are not caught by the compiler: the variable was correctly defined in both processes, but never propagated.

Compartmentalisation of gzip raises an important design question when working with capability mode: the changes were small, but non-trivial: is there a better way to apply sandboxing to applications most frequently used in pipelines? Seaborn has suggested one possibility: a Principle of Least Authority Shell (PLASH), in which the shell runs with ambient privilege and pipeline components are placed in sandboxes by the shell [21]. We have begun to explore this approach on Capsicum, but observe that the design tension exists here as well: `gzip`'s non-pipeline mode performs a number of application-specific operations requiring ambient privilege, and logic like this may be equally (if not more) awkward if placed in the shell. On the other hand, when operating purely in a pipeline, the PLASH approach offers the possibility of near-zero application modification.

Another area we are exploring is library self-compartmentalisation. With this approach, library code sandboxes portions of itself transparently to the host application. This approach motivated a number of our design choices, especially as relates to the process model: masking `SIGCHLD` delivery to the parent when using process descriptors allows libraries to avoid disturbing application state. This approach would allow video codec libraries to sandbox portions of themselves while executing in an unmodified web browser. However, library APIs are often not crafted for sandbox-friendliness: one reason we placed separation in `gzip` rather than `libz` is that `gzip` provided internal APIs based on file descriptors, whereas `libz` provided APIs based on buffers. Forwarding capabilities offers full UNIX I/O performance, whereas the cost of performing RPCs to transfer buffers between processes scales with file size. Likewise, historic vulnerabilities in `libjpeg` have largely centred on callbacks to applications rather than existing in isolation in the library; such callback interfaces require significant changes to run in an RPC environment.

## 4.4 Chromium

Google's Chromium web browser uses a multi-process architecture similar to a Capsicum logical application to improve robustness [18]. In this model, each tab is associated with a *renderer process* that performs the risky and complex task of rendering page contents through page parsing, image rendering, and JavaScript execution. More recent work on Chromium has integrated sandboxing techniques to improve resilience to malicious attacks rather than occasional instability; this has been done in various ways on different supported operating systems, as we will discuss in detail in Section 5.

The FreeBSD port of Chromium did not include sandboxing, and the sandboxing facilities provided as part of the similar Linux and Mac OS X ports bear little resemblance to Capsicum. However, the existing compartmentalisation meant that several critical tasks had already been performed:

- Chromium assumes that processes can be converted into sandboxes that limit new object access
- Certain services were already forwarded to renderers, such as font loading via passed file descriptors
- Shared memory is used to transfer output between renderers and the web browser
- Chromium contains RPC marshalling and passing code in all the required places

The only significant Capsicum change to the FreeBSD port of Chromium was to switch from System V shared memory (permitted in Linux sandboxes) to the POSIX shared memory code used in the Mac OS X port (capability-oriented and permitted in Capsicum's capability mode). Approximately 100 additional lines of code were required to introduce calls to `lc_limitfd` to limit access to file descriptors inherited by and passed to sandbox processes, such as Chromium data `pak` files, `stdio`, and `/dev/random`, font files, and to call `cap_enter`. This compares favourably with the 4.3 million lines of code in the Chromium source tree, but would not have been possible without existing sandbox support in the design. We believe it should be possible, without a significantly larger number of lines of code, to explore using the `libcapsicum` API directly.



| Operating system | Model | Line count | Description |
|---|---|---|---|
| Windows | ACLs | 22,350 | Windows ACLs and SIDs |
| Linux | `chroot` | 605 | `setuid` root helper sandboxes renderer |
| Mac OS X | Seatbelt | 560 | Path-based MAC sandbox |
| Linux | SELinux | 200 | Restricted sandbox type enforcement domain |
| Linux | `seccomp` | 11,301 | `seccomp` and userspace syscall wrapper |
| FreeBSD | Capsicum | 100 | Capsicum sandboxing using `cap_enter` |

Figure 12: Sandboxing mechanisms employed by Chromium.

# 5 Comparison of sandboxing technologies

We now compare Capsicum to existing sandbox mechanisms. Chromium provides an ideal context for this comparison, as it employs six sandboxing technologies (see Figure 12). Of these, the two are DAC-based, two MAC-based and two capability-based.

## 5.1 Windows ACLs and SIDs

On Windows, Chromium uses DAC to create sandboxes [18]. The unsuitability of inter-user protections for the intra-user context is demonstrated well: the model is both incomplete and unwieldy. Chromium uses Access Control Lists (ACLs) and Security Identifiers (SIDs) to sandbox renderers on Windows. Chromium creates a modified, reduced privilege, SID, which does not appear in the ACL of any object in the system, in effect running the renderer as an anonymous user.

However, objects which do not support ACLs are not protected by the sandbox. In some cases, additional precautions can be used, such as an alternate, invisible desktop to protect the user's GUI environment. However, unprotected objects include FAT filesystems on USB sticks and TCP/IP sockets: a sandbox cannot read user files directly, but it may be able to communicate with any server on the Internet or use a configured VPN! USB sticks present a significant concern, as they are frequently used for file sharing, backup, and protection from malware.

Many legitimate system calls are also denied to the sandboxed process. These calls are forwarded by the sandbox to a trusted process responsible for filtering and serving them. This forwarding comprises most of the 22,000 lines of code in the Windows sandbox module.

## 5.2 Linux chroot

Chromium's `suid` sandbox on Linux also attempts to create a privilege-free sandbox using legacy OS access control; the result is similarly porous, with the additional risk that OS privilege is required to create a sandbox.

In this model, access to the filesystem is limited to a directory via `chroot`: the directory becomes the sandbox's virtual root directory. Access to other namespaces, including System V shared memory (where the user's X window server can be contacted) and network access, is unconstrained, and great care must be taken to avoid leaking resources when entering the sandbox.

Furthermore, initiating `chroot` requires a `setuid` binary: a program that runs with full system privilege. While comparable to Capsicum's capability mode in terms of intent, this model suffers significant sandboxing weakness (for example, permitting full access to the System V shared memory as well as all operations on passed file descriptors), and comes at the cost of an additional setuid-root binary that runs with system privilege.

## 5.3 MAC OS X Seatbelt

On Mac OS X, Chromium uses a MAC-based framework for creating sandboxes. This allows Chromium to create a stronger sandbox than is possible with DAC, but the rights that are granted to render processes are still very broad, and security policy must be specified separately from the code that relies on it.

The Mac OS X *Seatbelt* sandbox system allows processes to be constrained according to a LISP-based policy language [1]. It uses the MAC Framework [27] to check application activities; Chromium uses three policies for different components, allowing access to filesystem elements such as font directories while restricting access to the global namespace.

Like other techniques, resources are acquired before constraints are imposed, so care must be taken to avoid leaking resources into the sandbox. Fine-grained filesystem constraints are possible, but other namespaces such as POSIX shared memory, are an all-or-nothing affair. The Seatbelt-based sandbox model is less verbose than other approaches, but like all MAC systems, security policy must be expressed separately from code. This can lead to inconsistencies and vulnerabilities.

## 5.4 SELinux

Chromium's MAC approach on Linux uses an SELinux Type Enforcement policy [12]. SELinux can be used

for very fine-grained rights assignment, but in practice, broad rights are conferred because fine-grained Type Enforcement policies are difficult to write and maintain. The requirement that an administrator be involved in defining new policy and applying new types to the file system is a significant inflexibility: application policies cannot adapt dynamically, as system privilege is required to reformulate policy and relabel objects.

The Fedora reference policy for Chromium creates a single SELinux dynamic domain, `chrome_sandbox_t`, which is shared by all sandboxes, risking potential interference between sandboxes. This domain is assigned broad rights, such as the ability to read all files in `/etc` and access to the terminal device. These broad policies are easier to craft than fine-grained ones, reducing the impact of the dual-coding problem, but are much less effective, allowing leakage between sandboxes and broad access to resources outside of the sandbox.

In contrast, Capsicum eliminates dual-coding by combining security policy with code in the application. This approach has benefits and drawbacks: while bugs can't arise due to potential inconsistency between policy and code, there is no longer an easily accessible specification of policy to which static analysis can be applied. This reinforces our belief that systems such as Type Enforcement and Capsicum are potentially complementary, serving differing niches in system security.

### 5.5 Linux seccomp

Linux provides an optionally-compiled capability mode-like facility called `seccomp`. Processes in `seccomp` mode are denied access to all system calls except `read`, `write`, and `exit`. At face value, this seems promising, but as OS infrastructure to support applications using `seccomp` is minimal, application writers must go to significant effort to use it.

In order to allow other system calls, Chromium constructs a process in which one thread executes in `seccomp` mode, and another "trusted" thread sharing the same address space has normal system call access. Chromium rewrites `glibc` and other library system call vectors to forward system calls to the trusted thread, where they are filtered in order to prevent access to inappropriate shared memory objects, opening files for write, etc. However, this default policy is, itself, quite weak, as read of any file system object is permitted.

The Chromium `seccomp` sandbox contains over a thousand lines of hand-crafted assembly to set up sandboxing, implement system call forwarding, and craft a basic security policy. Such code is a risky proposition: difficult to write and maintain, with any bugs likely leading to security vulnerabilities. The Capsicum approach is similar to that of `seccomp`, but by offering a richer set of services to sandboxes, as well as more granular delegation via capabilities, it is easier to use correctly.

## 6 Performance evaluation

Typical operating system security benchmarking is targeted at illustrating zero or near-zero overhead in the hopes of selling general applicability of the resulting technology. Our thrust is slightly different: we know that application authors who have already begun to adopt compartmentalisation are willing to accept significant overheads for mixed security return. Our goal is therefore to accomplish comparable performance with significantly improved security.

We evaluate performance in two ways: first, a set of micro-benchmarks establishing the overhead introduced by Capsicum's capability mode and capability primitives. As we are unable to measure any noticeable performance change in our adapted UNIX applications (`tcpdump` and `dhclient`) due to the extremely low cost of entering capability mode from an existing process, we then turn our attention to the performance of our `libcapsicum`-enhanced `gzip`.

All performance measurements have been performed on an 8-core Intel Xeon E5320 system running at 1.86GHz with 4GB of RAM, running either an unmodified FreeBSD 8-STABLE distribution synchronised to revision 201781 (2010-01-08) from the FreeBSD Subversion repository, or a synchronised 8-STABLE distribution with our capability enhancements.

### 6.1 System call performance

First, we consider system call performance through micro-benchmarking. Figure 13 summarises these results for various system calls on unmodified FreeBSD, and related capability operations in Capsicum. Figure 14 contains a table of benchmark timings. All micro-benchmarks were run by performing the target operation in a tight loop over an interval of at least 10 seconds, repeating for 10 iterations. Differences were computed using Student's t-test at $95\%$ confidence.

Our first concern is with the performance of capability creation, as compared to raw object creation and the closest UNIX operation, `dup`. We observe moderate, but expected, performance overheads for capability wrapping of existing file descriptors: the `cap_new` syscall is $50.7\% \pm 0.08\%$ slower than dup, or $539 \pm 0.8$ns slower in absolute terms.

Next, we consider the overhead of capability "unwrapping", which occurs on every descriptor operation. We compare the cost of some simple operations on raw file descriptors, to the same operations on a capability-wrapped version of the same file descriptor: writing a



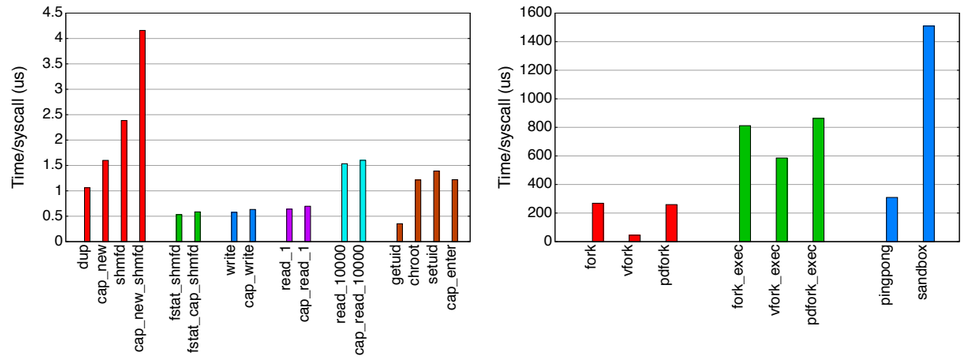



Figure 13: Capsicum system call performance compared to standard UNIX calls.

single byte to `/dev/null`, reading a single byte from `/dev/zero`; reading 10000 bytes from `/dev/zero`; and performing an `fstat` call on a shared memory file descriptor. In all cases we observe a small overhead of about $0.06\mu s$ when operating on the capability-wrapped file descriptor. This has the largest relative performance impact on `fstat` (since it does not perform I/O, simply inspecting descriptor state, it should thus experience the highest overhead of any system call which requires unwrapping). Even in this case the overhead is relatively low: $10.2\% \pm 0.5\%$.

### 6.2 Sandbox creation

Capsicum supports ways to create a sandbox: directly invoking `cap_enter` to convert an existing process into a sandbox, inheriting all current capability lists and memory contents, and the `libcapsicum` sandbox API, which creates a new process with a flushed capability list.

`cap_enter` performs similarly to `chroot`, used by many existing compartmentalised applications to restrict file system access. However, `cap_enter` out-performs `setuid` as it does not need to modify resource limits. As most sandboxes `chroot` and set the UID, entering a capability mode sandbox is roughly twice as fast as a traditional UNIX sandbox. This suggests that the overhead of adding capability mode support to an application with existing compartmentalisation will be negligible, and replacing existing sandboxing with `cap_enter` may even marginally improve performance.

Creating a new sandbox process and replacing its address space using `execve` is an expensive operation. Micro-benchmarks indicate that the cost of `fork` is three orders of magnitude greater than manipulating the process credential, and adding `execve` or even a single instance of message passing increases that cost further. We also found that additional dynamically linked library dependencies (`libcapsicum` and its dependency on `libsbuf`) impose an additional $9\%$ cost to the `fork` syscall, presumably due to the additional virtual memory mappings being copied to the child process. This overhead is not present on `vfork` which we plan to use in `libcapsicum` in the future. Creating, exchanging an RPC with, and destroying a single sandbox (the "sandbox" label in Figure 13(b)) has a cost of about 1.5ms, significantly higher than its subset components.

### 6.3 gzip performance

While the performance cost of `cap_enter` is negligible compared to other activity, the cost of multi-process sandbox creation (already taken by `dhclient` and Chromium due to existing sandboxing) is significant.

To measure the cost of process sandbox creation, we timed `gzip` compressing files of various sizes. Since the additional overheads of sandbox creation are purely at startup, we expect to see a constant-time overhead to the capability-enhanced version of `gzip`, with identical linear scaling of compression performance with input file size. Files were pre-generated on a memory disk by reading a constant-entropy data source: `/dev/zero` for perfectly compressible data, `/dev/random` for perfectly incompressible data, and base 64-encoded `/dev/random` for a moderate high entropy data source, with about $24\%$ compression after gzipping. Using a data source with approximately constant entropy per bit minimises variation in overall `gzip` performance due to changes in compressor performance as files of different sizes are sampled. The list of files was piped to `xargs -n 1 gzip -c > /dev/null`, which sequentially invokes a new `gzip`

| Benchmark | Time/operation | Difference | % difference |
|---|---|---|---|
| dup | $1.061 \pm 0.000\mu s$ | - | - |
| cap_new | $1.600 \pm 0.001\mu s$ | $0.539 \pm 0.001\mu s$ | $50.7\% \pm 0.08\%$ |
| shmfd | $2.385 \pm 0.000\mu s$ | - | - |
| cap_new_shmfd | $4.159 \pm 0.007\mu s$ | $1.77 \pm 0.004\mu s$ | $74.4\% \pm 0.181\%$ |
| fstat_shmfd | $0.532 \pm 0.001\mu s$ | - | - |
| fstat_cap_shmfd | $0.586 \pm 0.004\mu s$ | $0.054 \pm 0.003\mu s$ | $10.2\% \pm 0.506\%$ |
| read_1 | $0.640 \pm 0.000\mu s$ | - | - |
| cap_read_1 | $0.697 \pm 0.001\mu s$ | $0.057 \pm 0.001\mu s$ | $8.93\% \pm 0.143\%$ |
| read_10000 | $1.534 \pm 0.000\mu s$ | - | - |
| cap_read_10000 | $1.601 \pm 0.003\mu s$ | $0.067 \pm 0.002\mu s$ | $4.40\% \pm 0.139\%$ |
| write | $0.576 \pm 0.000\mu s$ | - | - |
| cap_write | $0.634 \pm 0.002\mu s$ | $0.058 \pm 0.001\mu s$ | $10.0\% \pm 0.241\%$ |
| cap_enter | $1.220 \pm 0.000\mu s$ | - | - |
| getuid | $0.353 \pm 0.001\mu s$ | $-0.867 \pm 0.001\mu s$ | $-71.0\% \pm 0.067\%$ |
| chroot | $1.214 \pm 0.000\mu s$ | $-0.006 \pm 0.000\mu s$ | $-0.458\% \pm 0.023\%$ |
| setuid | $1.390 \pm 0.001\mu s$ | $0.170 \pm 0.001\mu s$ | $14.0\% \pm 0.054\%$ |
| fork | $268.934 \pm 0.319\mu s$ | - | - |
| vfork | $44.548 \pm 0.067\mu s$ | $-224.3 \pm 0.217\mu s$ | $-83.4\% \pm 0.081\%$ |
| pdfork | $259.359 \pm 0.118\mu s$ | $-9.58 \pm 0.324\mu s$ | $-3.56\% \pm 0.120\%$ |
| pingpong | $309.387 \pm 1.588\mu s$ | $40.5 \pm 1.08\mu s$ | $15.0\% \pm 0.400\%$ |
| fork_exec | $811.993 \pm 2.849\mu s$ | - | - |
| vfork_exec | $585.830 \pm 1.635\mu s$ | $-226.2 \pm 2.183\mu s$ | $-27.9\% \pm 0.269\%$ |
| pdfork_exec | $862.823 \pm 0.554\mu s$ | $50.8 \pm 2.83\mu s$ | $6.26\% \pm 0.348\%$ |
| sandbox | $1509.258 \pm 3.016\mu s$ | $697.3 \pm 2.78\mu s$ | $85.9\% \pm 0.339\%$ |

Figure 14: Micro-benchmark results for various system calls and functions, grouped by category.

compression process with a single file argument, and discards the compressed output. Sufficiently many input files were generated to provide at least 10 seconds of repeated `gzip` invocations, and the overall run-time measured. I/O overhead was minimised by staging files on a memory disk. The use of `xargs` to repeatedly invoke `gzip` provides a tight loop that minimising the time between `xargs`' successive `vfork` and `exec` calls of `gzip`. Each measurement was repeated 5 times and averaged.

Benchmarking `gzip` shows high initial overhead, when compressing single-byte files, but also that the approach in which file descriptors are wrapped in capabilities and delegated rather than using pure message passing, leads to asymptotically identical behaviour as file size increases and run-time cost are dominated by compression workload, which is unaffected by Capsicum. We find that the overhead of launching a sandboxed `gzip` is $2.37 \pm 0.01$ ms, independent of the type of compression stream. For many workloads, this one-off performance cost is negligible, or can be amortised by passing multiple files to the same `gzip` invocation.

## 7 Future work

Capsicum provides an effective platform for capability work on UNIX platforms. However, further research and development are required to bring this project to fruition.

We believe further refinement of the Capsicum primitives would be useful. Performance could be improved for sandbox creation, perhaps employing an Capsicum-centric version of the S-thread primitive proposed by Bittau. Further, a "logical application" OS construct might improve termination properties.

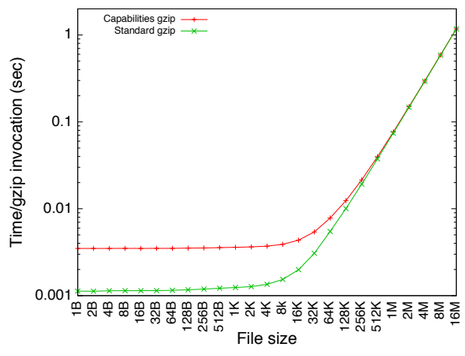



Figure 15: Run time per `gzip` invocation against random data, with varying file sizes; performance of the two versions come within 5% of one another at around a 512K.

Another area for research is in integrating user interfaces and OS security; Shapiro has proposed that capability-centered window systems are a natural extension to capability operating systems. Improving the mapping of application security constructs into OS sandboxes would also significantly improve the security of Chromium, which currently does not consistently assign web security domains to sandboxes. It is in the context of windowing systems that we have found capability delegation most valuable: by driving delegation with UI behaviors, such as Powerboxes (file dialogues running with ambient authority) and drag-and-drop, Capsicum can support gesture-based access control research.

Finally, it is clear that the single largest problem with Capsicum and other privilege separation approaches is programmability: converting local development into de facto distributed development adds significant complexity to code authoring, debugging, and maintenance. Likewise, aligning security separation with application separation is a key challenge: how does the programmer identify and implement compartmentalisations that offer real security benefits, and determine that they've done so correctly? Further research in these areas is critical if systems such as Capsicum are to be used to mitigate security vulnerabilities through process-based compartmentalisation on a large scale.

## 8 Related work

In 1975, Saltzer and Schroeder documented a vocabulary for operating system security based on on-going work on MULTICS [19]. They described the concepts of capabilities and access control lists, and observed that in practice, systems combine the two approaches in order to offer a blend of control and performance. Thirty-five years of research have explored these and other security concepts, but the themes remain topical.

### 8.1 Discretionary and Mandatory Access Control

The principle of discretionary access control (DAC) is that users control protections on objects they own. While DAC remains relevant in multi-user server environments, the advent of personal computers and mobile phones has revealed its weakness: on a single-user computer, all eggs are in one basket. Section 5.1 demonstrates the difficulty of using DAC for malicious code containment.

Mandatory access control systemically enforce policies representing the interests of system implementers and administrators. Information flow policies tag subjects and objects in the system with confidentiality and integrity labels—fixed rules prevent reads or writes that allowing information leakage. Multi-Level Security (MLS), formalised as Bell-LaPadula (BLP), protects confidential information from unauthorised release [3]. MLS's logical dual, the Biba integrity policy, implements a similar scheme protecting integrity, and can be used to protect Trusted Computing Bases (TCBs) [4].

MAC policies are robust against the problem of *confused deputies*, authorised individuals or processes who can be tricked into revealing confidential information. In practice, however, these policies are highly inflexible, requiring administrative intervention to change, which precludes browsers creating isolated and ephemeral sandboxes "on demand" for each web site that is visited.

Type Enforcement (TE) in LOCK [20] and, later, SELinux [12] and SEBSD [25], offers greater flexibility by allowing arbitrary labels to be assigned to subjects (domains) and objects (types), and a set of rules to control their interactions. As demonstrated in Section 5.4, requiring administrative intervention and the lack of a facility for ephemeral sandboxes limits applicability for applications such as Chromium: policy, by design, cannot be modified by users or software authors. Extreme granularity of control is under-exploited, or perhaps even discourages, highly granular protection—for example, the Chromium SELinux policy conflates different sandboxes allowing undesirable interference.

### 8.2 Capability systems, micro-kernels, and compartmentalisation

The development of capability systems has been tied to mandatory access control since conception, as capabilities were considered the primitive of choice for mediation in trusted systems. Neumann et al's Provably Secure Operating System (PSOS) [16], and successor LOCK, propose a tight integration of the two models, with the later refinement that MAC allows revocation of capabilities in order to enforce the *-property [20].

Despite experimental hardware such as Wilkes' CAP computer [28], the eventual dominance of general-purpose virtual memory as the nearest approximation of hardware capabilities lead to exploration of object-capability systems and micro-kernel design. Systems such as Mach [2], and later L4 [11], epitomise this approach, exploring successively greater extraction of historic kernel components into separate tasks. Trusted operating system research built on this trend through projects blending mandatory access control with micro-kernels, such as Trusted Mach [6], DTMach [22] and FLASK [24]. Micro-kernels have, however, been largely rejected by commodity OS vendors in favour of higher-performance monolithic kernels.

MAC has spread, without the benefits of micro-kernel-enforced reference monitors, to commodity UNIX sys-

tems in the form of SELinux [12]. Operating system capabilities, another key security element to micro-kernel systems, have not seen wide deployment; however, research has continued in the form of EROS [23] (now CapROS), inspired by KEYKOS [9].

OpenSSH privilege separation [17] and Privman [10] rekindled interest in micro-kernel-like compartmentalisation projects, such as the Chromium web browser [18] and Capsicum's logical applications. In fact, large application suites compare formidably with the size and complexity of monolithic kernels: the FreeBSD kernel is composed of 3.8 million lines of C, whereas Chromium and WebKit come to a total of 4.1 million lines of C++. How best to decompose monolithic applications remains an open research question; Bittau's Wedge offers a promising avenue of research in automated identification of software boundaries through dynamic analysis [5].

Seaborn and Hand have explored application compartmentalisation on UNIX through capability-centric Plash [21], and Xen [15], respectively. Plash offers an intriguing blend of UNIX semantics with capability security by providing POSIX APIs over capabilities, but is forced to rely on the same weak UNIX primitives analysed in Section 5. Supporting Plash on stronger Capsicum foundations would offer greater application compatibility to Capsicum users. Hand's approach suffers from similar issues to `seccomp`, in that the run-time environment for sandboxes is functionality-poor. Garfinkel's Ostia [7] also considers a delegation-centric approach, but focuses on providing sandboxing as an extension, rather than a core OS facility.

A final branch of capability-centric research is capability programming languages. Java and the JVM have offered a vision of capability-oriented programming: a language run-time in which references and byte code verification don't just provide implementation hiding, but also allow application structure to be mapped directly to protection policies [8]. More specific capability-oriented efforts are E [13], the foundation for Capdesk and the DARPA Browser [26], and Caja, a capability subset of the JavaScript language [14].

## 9 Conclusion

We have described Capsicum, a practical capabilities extension to the POSIX API, and a prototype based on FreeBSD, planned for inclusion in FreeBSD 9.0. Our goal has been to address the needs of application authors who are already experimenting with sandboxing, but find themselves building on sand when it comes to effective containment techniques. We have discussed our design choices, contrasting approaches from research capability systems, as well as commodity access control and sandboxing technologies, but ultimately leading to a new approach. Capsicum lends itself to adoption by blending immediate security improvements to current applications with the long-term prospects of a more capability-oriented future. We illustrate this through adaptations of widely-used applications, from the simple `gzip` to Google's highly-complex Chromium web browser, showing how firm OS foundations make the job of application writers easier. Finally, security and performance analyses show that improved security is not without cost, but that the point we have selected on a spectrum of possible designs improves on the state of the art.

## 10 Acknowledgments

The authors wish to gratefully acknowledge our sponsors, including Google, Inc, the Rothermere Foundation, and the Natural Sciences and Engineering Research Council of Canada. We would further like to thank Mark Seaborn, Andrew Moore, Joseph Bonneau, Saar Drimer, Bjoern Zeeb, Andrew Lewis, Heradon Douglas, Steve Bellovin, and our anonymous reviewers for helpful feedback on our APIs, prototype, and paper, and Sprewell for his contributions to the Chromium FreeBSD port.

## 11 Availability

Capsicum, as well as our extensions to the Chromium web browser are available under a BSD license; more information may be found at:

`http://www.cl.cam.ac.uk/research/security/capsicum/`

A technical report with additional details is forthcoming.

## References

[1] The Chromium Project: Design Documents: OS X Sandboxing Design. `http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design`.

[2] ACETTA, M. J., BARON, R., BOLOWSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: a new kernel foundation for unix development. In *Proceedings of the USENIX 1986 Summer Conference* (July 1986), pp. 93–112.

[3] BELL, D. E., AND LAPADULA, L. J. Secure computer systems: Mathematical foundations. Tech. Rep. 2547, MITRE Corp., March 1973.

[4] BIBA, K. J. Integrity considerations for secure computer systems. Tech. rep., MITRE Corp., April 1977.

[5] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), pp. 309–322.

[6] BRANSTAD, M., AND LANDAUER, J. Assurance for the Trusted Mach operating system. *Computer Assurance, 1989. COMPASS '89, 'Systems Integrity, Software Safety and Process Security', Proceedings of the Fourth Annual Conference on* (1989), 103–108.

[7] GARFINKEL, T., PFA, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proc. Internet Society 2003* (2003).

[8] GONG, L., MUELLER, M., PRAFULLCHANDRA, H., AND SCHEMERS, R. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*.

[9] HARDY, N. KeyKOS architecture. *SIGOPS Operating Systems Review 19*, 4 (Oct 1985).

[10] KILPATRICK, D. Privman: A Library for Partitioning Applications. In *Proceedings of USENIX Annual Technical Conference* (2003), pp. 273–284.

[11] LIEDTKE, J. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)* (Copper Mountain Resort, CO, Dec. 1995).

[12] LOSCOCCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the Linux operating system. *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference table of contents* (2001), 29–42.

[13] MILLER, M. S. The e language. `http://www.erights.org/`.

[14] MILLER, M. S., SAMUEL, M., LAURIE, B., AWAD, I., AND STAY, M. Caja: Safe active content in sanitized javascript, May 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[15] MURRAY, D. G., AND HAND, S. Privilege Separation Made Easy. In *Proceedings of the ACM SIGOPS European Workshop on System Security (EUROSEC)* (2008), pp. 40–46.

[16] NEUMANN, P. G., BOYER, R. S., GEIERTAG, R. J., LEVITT, K. N., AND ROBINSON, L. A provably secure operating system: The system, its applications, and proofs, second edition. Tech. Rep. Report CSL-116, Computer Science Laboratory, SRI International, May 1980.

[17] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium* (2003).

[18] REIS, C., AND GRIBBLE, S. D. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems* (New York, NY, USA, 2009), ACM, pp. 219–232.

[19] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. In *Communications of the ACM* (July 1974), vol. 17.

[20] SAMI SAYDJARI, O. Lock: an historical perspective. In *Proceeedings of the 18th Annual Computer Security Applications Conference* (2002), IEEE Computer Society.

[21] SEABORN, M. Plash: tools for practical least privilege, 2010. `http://plash.beasts.org/`.

[22] SEBES, E. J. Overview of the architecture of Distributed Trusted Mach. *Proceedings of the USENIX Mach Symposium: November* (1991), 20–22.

[23] SHAPIRO, J., SMITH, J., AND FARBER, D. EROS: a fast capability system. *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles* (Dec 1999).

[24] SPENCER, R., SMALLEY, S., LOSCOCCO, P., HIBLER, M., ANDERSON, D., AND LEPREAU, J. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. 8th USENIX Security Symposium* (August 1999).

[25] VANCE, C., AND WATSON, R. Security Enhanced BSD. *Network Associates Laboratories* (2003).

[26] WAGNER, D., AND TRIBBLE, D. A security analysis of the combex darpabrowser architecture, March 2002. `http://www.combex.com/papers/darpa-review/security-review.pdf`.

[27] WATSON, R., FELDMAN, B., MIGUS, A., AND VANCE, C. Design and Implementation of the TrustedBSD MAC Framework. In *Proc. Third DARPA Information Survivability Conference and Exhibition (DISCEX), IEEE* (April 2003).

[28] WILKES, M. V., AND NEEDHAM, R. M. *The Cambridge CAP computer and its operating system (Operating and programming systems series)*. Elsevier North-Holland, Inc., Amsterdam, The Netherlands, 1979.

# Structuring Protocol Implementations to Protect Sensitive Data

Petr Marchenko and Brad Karp
University College London, Gower Street, London WC1E 6BT, UK
{p.marchenko,bkarp}@cs.ucl.ac.uk

## Abstract

In a bid to limit the harm caused by ubiquitous remotely exploitable software vulnerabilities, the computer systems security community has proposed primitives to allow execution of application code with reduced privilege. In this paper, we identify and address the vital and largely unexamined problem of *how to structure* implementations of cryptographic protocols to protect sensitive data despite exploits. As evidence that this problem is poorly understood, we first identify two attacks that lead to disclosure of sensitive data in two published state-of-the-art designs for exploit-resistant cryptographic protocol implementations: privilege-separated OpenSSH, and the HiStar/DStar DIFC-based SSL web server. We then describe how to structure protocol implementations on UNIX- and DIFC-based systems to defend against these two attacks and protect sensitive information from disclosure. We demonstrate the practicality and generality of this approach by applying it to protect sensitive data in the implementations of both the server and client sides of OpenSSH and of the OpenSSL library.

## 1 Introduction

Cryptographic protocols are entrusted to preserve the integrity and secrecy of sensitive data as it traverses a network. While these protocols incorporate strong mechanisms to defend against in-network eavesdropping and modification of data in transit, such protocols function in today's distributed systems only as imperfect, human-written software. Clearly, the desired outcome for secure system designers implementing a secure data transfer protocol like SSH [13] or SSL/TLS [4] is *end-to-end* integrity and secrecy for sensitive data, despite not only in-network threats, but also threats that may arise from the behavior of the protocol implementation(s) at the ends of the wire. The dismal past two decades of remotely exploitable vulnerabilities in software deployed widely on network-attached hosts are thus real cause for alarm—even if the abstract design of a cryptographic protocol is correct, the protocol's very implementation is a worryingly weak link in achieving end-to-end security goals.

In the quest for a lasting end-to-end defense for sensitive data against disclosure or corruption by a remote attacker, whatever vulnerabilities and exploits come to light in the future, the systems research community has in recent years sought to put the venerable *principle of least privilege* [10] into better practice in the software running on network-connected servers. This design tenet dictates that the programmer should partition his code into compartments, each of which executes a portion of the program with minimal privilege necessary to carry out its function. Here, privilege corresponds to access rights for system resources: to read or write the filesystem, memory, or network, to invoke a system call, *&c.* In the context of exploitable vulnerabilities and sensitive information, least privilege amounts to designing an application with the expectation that exploits will occur, but limiting the harm that they may cause by restricting the actions that an attacker may take post-exploit.

Early work [5, 9] explored how to minimize privilege on compartments instantiated as standard UNIX processes. More recently, the community has devoted considerable effort to providing various operating system primitives intended to make it easier for programmers to adhere to the principle of least privilege. These primitives range from operating system support for decentralized information flow control (DIFC) [6, 12, 14, 15], which limits the privileges of any compartment exposed to sensitive information, to process-like primitives that lessen the likelihood of accidental propagation of privileges between compartments against the programmer's intent [2].

While these results all represent important advances over the prior state of the art, we believe that proposals to date for new primitives to encourage programmers' adherence to least privilege largely ignore a central, vital question: *how should a programmer structure code and limit privilege to prevent disclosure or corruption of sensitive data by an attacker who can exploit a vulnerability?* Regardless of the primitives used, this daunting question looms. To their credit, the proposers of these primitives present examples of how to structure application code to use them. But these examples are typically offered as existential evidence that the primitives themselves are useful; no guidance or principles are offered for how one may structure an application's code to use the primitives and robustly provide the desired end-to-end secrecy and/or integrity guarantees.

Moreover, the structures of these example applications are complex, as they are typically split into many compartments. To wit, the OKWS web server spreads

its code among at least 5 compartments (processes) [5], the sthread-partitioned Apache/SSL web server consists of 9 compartments (sthreads and callgates) [2], and the HiStar/DStar-labeled Apache/SSL web server consists of 7 compartments (processes) [15]. Each application's many compartments are configured with different privileges and labels, respectively, and interconnected in complex patterns. Structuring code to use these primitives appears difficult. Indeed, as we show in Section 3, even highly security-conscious programmers using state-of-the-art techniques [9, 15] have not adequately considered how to defend cryptographic protocol implementations from exploit-based attacks.

In this paper, we offer a practical improvement over the status quo: principles to guide programmers in structuring cryptographic protocol implementations so as to robustly protect sensitive user data *end-to-end,* including in cases where a remote attacker exploits untrusted application code. Our contributions include:

- We define two general classes of attack on cryptographic protocol implementations: *session key disclosure attacks* and *oracle attacks.* We demonstrate that two state-of-the-art cryptographic protocol implementations, one in privilege-separated OpenSSH [9] and the other in a DIFC-labeled Apache/SSL web server [15], are vulnerable to these attacks.
- We provide protocol-agnostic principles for structuring cryptographic protocol implementations to protect sensitive data against disclosure and corruption when an exploitable vulnerability is present in code that processes network input.
- As evidence of the practicality and generality of these principles, we present restructured implementations of the OpenSSH server and client and of the OpenSSL library that limit privilege so as to protect users' sensitive data from an adversary who can remotely exploit the implementation. This restructured OpenSSL library can act as a drop-in replacement for the stock library, bringing robustness against these attacks to a wide range of SSL-enabled applications.

## 2 Background

We now summarize the state of the art in protecting sensitive data in network server software. The two main approaches in use are privilege separation and decentralized information flow control (DIFC).

### 2.1 Privilege Separation with Processes

In a monolithic application, in which all code executes in a single compartment (under UNIX or Linux, a process), all instructions execute with full privilege. Thus, an exploit of a vulnerability may result in disclosure of sensitive data, and more generally, grants the full privilege held by the application to any code injected by the attacker. Privilege separation [9] has proven effective in mitigating these threats. This technique follows from the observation that an application need not execute individual operations with the union of all privileges needed by all operations during the application's entire lifetime. Many vulnerability-prone operations, such as parsing, do not require access to sensitive information or the filesystem. If we partition a monolithic application into compartments and restrict some compartments' privileges, an exploit in an unprivileged compartment will not be able to disclose or corrupt sensitive information to which it does not have access. Code that runs in privileged compartments, however, must be carefully audited to protect the sensitive data it can access.

The privilege-separated OpenSSH server [9] divides the server's code into separate standard UNIX/Linux processes. This partitioning includes a network-facing unprivileged process that performs key exchange and authentication protocols, and a privileged monitor process running as `root` that exports an interface to the unprivileged process to allow invocation of privileged operations, such as signing with the server's private key, verifying user credentials, *&c.*

This structure is intended to deny the attacker execution of code with `root` privilege on the server; the attacker only interacts directly with the unprivileged process. Provos *et al.* state that "programming errors occurring in the unprivileged parts can no longer be abused to gain unauthorized privileges" [9]. This claim holds because the unprivileged process executes with restricted file system access (enforced with a `chroot` system call), and with unused user and group IDs of `nobody`, which prevent it from tampering with other processes.

The SELinux security extensions to Linux [7], which post-date Provos *et al.*'s work, allow enforcement of flexible mandatory access control policies specified by a system administrator. These policies support finer-grained restriction of a process's privileges than under stock Linux, primarily by checking system call invocations in the kernel against a per-process access control list. We employ these extensions in our cryptographic protocol implementations for OpenSSH and OpenSSL.

### 2.2 DIFC

Decentralized information flow control (DIFC), as implemented in the research prototype operating systems Asbestos [12] and HiStar [14], and retrofitted to Linux in Flume [6], offers a different approach to limiting privilege within applications. In these systems, a programmer expresses an information flow policy by labeling data according to its sensitivity level. Should an unprivileged compartment access data labeled as sensitive, it becomes tainted, and at run-time, the operating system prevents it from communicating with compartments tainted with



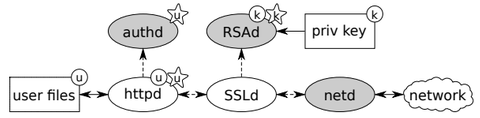



**Figure 1**: HiStar-labeled SSL web server. We omit *SSLd*'s and *netd*'s labels in the interest of brevity.

lower levels of sensitivity, or with the network or console. This way, an unprivileged compartment cannot convey sensitive data out of the application. To allow output, *trusted compartments* perform privileged operations on sensitive data: they *own* sensitive labels, and are thus allowed by the operating system to *declassify* sensitive information, stripping it of its sensitivity label(s).

Building on these DIFC primitives, Zeldovich *et al.* present a state-of-the-art privilege-separated SSL web server [15], shown in slightly simplified form in Figure 1. Ovals represent code: shaded ovals are trusted, privileged compartments, while white ovals are untrusted compartments. A dashed arrow between compartments *A* and *B* indicates that *A* may invoke an operation in *B* with arguments and retrieve the result. Boxes represent sensitive data. A solid arrow from data to a compartment denotes that the compartment may read that data; an arrow in the reverse direction denotes write access. Circles annotating data items and compartments indicate labels; in the latter case, a compartment is tainted with the label in question. Finally, a label within a star denotes that a compartment owns that label (and may declassify data labeled with it).

The HiStar-labeled SSL web server is partitioned into several untrusted compartments to limit the effect of a compromise of any single one. The major compartments are per-connection *SSLd*, per-connection *httpd*, and shared *RSAd* daemons. *SSLd* handles a client's SSL connection and performs key exchange, server authentication, encryption and decryption. *httpd* processes cleartext HTTP requests; it uses *SSLd* to decrypt requests and encrypt replies. *httpd* can obtain ownership of a user's label by authenticating with the trusted *authd* daemon. Label ownership allows *httpd* to read the user's data and declassify it for transfer over the network. The trusted *netd* serves as a barrier between the application and the network. It passes only declassified data (with no label) to the network.

## 3 Attacks on Protocol Implementations

The designers of cryptographic protocols like SSH and SSL aim to provide end-to-end confidentiality and integrity for users' data transferred during a session. When applied correctly, both privilege separation and DIFC can ensure that exploits of unprivileged compartments in a protocol's implementation will not lead to violations of these properties. In this section, we present two attacks

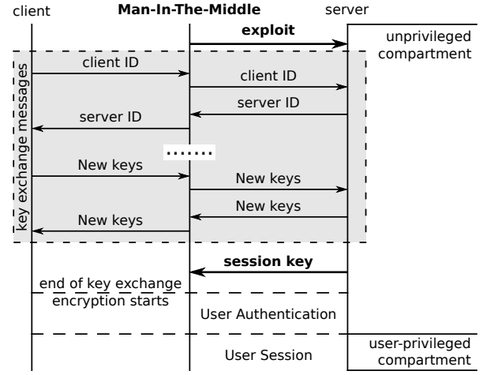



**Figure 2**: Session key disclosure attack against privilege-separated OpenSSH server.

that violate the confidentiality and integrity of sensitive user data in two state-of-the-art privilege-separated systems: one in privilege-separated OpenSSH, and one in a HiStar-labeled Apache-derived SSL web server.[1]

### 3.1 Session Key Disclosure Attack

The partitioning goal stated by the designers of privilege-separated OpenSSH was to prevent attackers' executing code with `root` privilege. However, as we will see, that goal is not sufficient to preserve the confidentiality and integrity of the *user's* sensitive data.

In prior work [2], we described an active man-in-the-middle attack against an SSL-enabled Apache Web server. This attack, which we term the *session key disclosure attack* (SKD attack), is also valid against a privilege-separated OpenSSH server. While in prior work we only discussed this attack against an SSL implementation, we now demonstrate that this attack applies against any protocol in which the two parties share a symmetric secret key.

In the SKD attack, an active man in the middle compromises an unprivileged compartment on the server, discloses the user's session key, and can then decrypt the sensitive data transmitted during the session. This attack succeeds because the unprivileged compartment responsible for key exchange and server authentication can read the session key shared between the server and client. We illustrate the SKD attack on Diffie-Hellman (DH) key exchange in OpenSSH in Figure 2. Here an unprivileged compartment processes key exchange messages and invokes a privileged monitor to sign a session ID with the server's private key (the privileged monitor is not shown in the figure). The user-privileged compartment executes with the authenticated user's UID and provides a remotely accessible shell.

The attacker begins by exploiting the server's unprivi-

leged compartment. He relays all key exchange messages to and from a legitimate user. The server and user compute a shared session key, which the attacker's injected code sends the attacker from the compromised compartment. After user authentication, the user transmits sensitive data encrypted with the compromised session key. Using the session key, the attacker can reveal the user's sensitive data, as well as inject her own commands and obtain further sensitive information stored on the server. Moreover, the session key also provides secrecy for user authentication, so the password of a client using password authentication will be compromised.

The state-of-the-art, HiStar-labeled SSL web server [15] aims to safeguard users' sensitive data from disclosure to other users. We note with interest that because the designers of this cryptographic protocol implementation did not consider the SKD attack when structuring their code, this server is vulnerable to the SKD attack in the same way that the privilege-separated OpenSSH server is. Specifically, the untrusted *SSLd* compartment computes a session key for a user's connection, but if an active man-in-the-middle attacker compromises this compartment, she may disclose the session key.

## 3.2 Oracle Attack

Next, consider the HiStar-labeled SSL web server shown in Figure 1. Depending on the key exchange protocol in use, *RSAd* signs either the ephemeral RSA key or the public DH components supplied by the untrusted *SSLd* with the server's permanent private key. This signature authenticates the server to the client. It is possible, however, to abuse the signing operation exported by *RSAd*. Although a compromised *SSLd* cannot directly read the private key, it can sign any data chosen by the attacker; the attacker controls the *SSLd* compartment, and can invoke *RSAd* with any arguments she chooses. Thus, the attacker can use a compromised *SSLd* to produce valid signatures using the server's identity. This example demonstrates that simply putting sensitive data beyond direct reach of untrusted code does not provide sufficient isolation.

We name such attacks against a cryptographic protocol's partitioning *oracle attacks.* Any trusted compartment or sequence of trusted compartments isolating sensitive data and exporting privileged operations to untrusted code can be an oracle. An oracle takes untrusted input from untrusted code and returns the result of a privileged operation. An attacker can obtain sensitive information by invoking the trusted compartment with appropriately chosen inputs. *SSLd* is meant only to pass *RSAd* an ephemeral key or the DH components for *its own current session* for signing. But if an active man-in-the-middle attacker compromises *SSLd*, she can sign arbitrary keys and DH components and present them to other users, and so impersonate the server.

We have further identified oracle structures in the "baseline" privilege-separated OpenSSH server [9]. The trusted monitor process exposes a private key-signing operation to the unprivileged compartment for authentication of the server during key exchange. The unprivileged compartment thus has an oracle for the server's private key, and an attacker who compromises that compartment can impersonate the OpenSSH server, just as was described for the SSL web server above.

While studying the SSH and SSL/TLS protocols, we identified further oracle attacks. Digital signatures suffer not only from signing oracles, but also verification oracles, in which an attacker can force successful signature verification by supplying chosen inputs to a trusted compartment performing this privileged operation. There also exists an oracle where an attacker forces a set of trusted compartments generating a session key to produce the same key used in a past user's session; we name this oracle a *deterministic session key oracle*. Forcing reuse of a session key allows an attacker to replay messages from a past session. (This particular threat exists in SSL's RSA key exchange protocol.) Finally, *encryption and decryption oracles* may allow an attacker to encrypt arbitrary data and decrypt confidential messages.

## 3.3 Discussion

The SKD and oracle attacks are *independent* of the low-level system primitive used to limit privilege; they appear equally in applications built with privilege separation and DIFC. These attacks are made possible by *weakly structured cryptographic protocol implementations.* The implementation of a cryptographic protocol should guarantee the same properties provided in the middle of the network: data confidentiality, data integrity, and robust authentication of the peers, even if untrusted compartments in its implementation are compromised. Avoiding SKD and oracle attacks requires subtle structuring of the implementation of a cryptographic protocol.

The SKD and oracle attacks target *building blocks* of cryptographic protocols. Risk of an SKD attack exists in many cases where a session key and key exchange protocol are used. Similarly, oracle attacks are associated with basic cryptographic operations such as encryption, decryption, signing, signature verification, message authentication, *&c*.

We next propose guiding principles for defense against SKD and oracle attacks. Just as these attacks arise in building blocks for cryptographic protocols, these principles concern how to implement these building blocks safely. We thus believe both the attacks and defenses apply to many cryptographic protocols.[2]



# 4 Principles for Partitioning

In this section, we define principles to guide the programmer when partitioning an implementation of a cryptographic protocol into reduced-privilege compartments. These principles allow preserving the key end-to-end security properties of the protocol, even when untrusted compartments are compromised. Our principles are agnostic to the underlying privilege-enforcement mechanism. Thus, they may be applied in DIFC-based systems, in privilege-separated systems based on Linux processes, and in other systems. They apply both to the client and server sides of cryptographic protocols.

Throughout, we assume that an attacker can compromise untrusted code and execute arbitrary code in its compartment, though only with the privileges allowed in that compartment. In this threat model, if an untrusted compartment acquires sensitive information or an attacker compromises a privileged compartment, we presume she obtains sensitive information.

## 4.1 Two-Barrier, Three-Stage Partitioning

A cryptographic protocol typically shares a symmetric secret key between two communicating parties, used to compute message authentication codes (MACs) and to encrypt data. A key exchange protocol confidentially shares this symmetric key. In addition, in some applications, the cryptographic protocol must authenticate peers to each other. Any authentication method that does not rely on transferring sensitive data, such as public key authentication, may be performed during the key exchange protocol, before a session-key-encrypted channel has been established. The SSL/TLS protocol fits this model [4]. In contrast, password-based authentication, *e.g.,* as supported by SSH [13], sends sensitive data over the network, and must therefore only authenticate after the session-key-MACed and -encrypted channel has been established. After authentication, an application is assured of the remote principal's identity, and can grant the remote principal access to locally stored sensitive data.

We distinguish two attack models. The first is that of the SKD attack described in Section 3.1, where a man-in-the-middle attacker exploits a vulnerability in a client or server application to obtain the peers' session key. The second attack model is that of an *impersonation attack,* where an attacker exploits an endpoint and subverts authentication in order to impersonate one of the peers.

In order to prevent these attacks, a partitioned application should implement structures that we term a *session key barrier* and a *user privilege barrier.* These divide an application into three stages, as shown in Figure 3. The first such stage, the *session key negotiation stage*, performs the key exchange protocol. The second stage, the *pre-authenticated stage*, conducts peer authentication. Finally, the *post-authenticated stage* processes user requests. Within each stage, one untrusted compartment handles network input and executes without privileges to read or write sensitive data, while multiple trusted compartments execute with privilege to access sensitive data. These trusted compartments export any necessary privileged operations to the untrusted compartment.

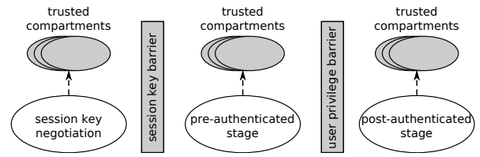



**Figure 3**: Barriers and stages in protocol partitioning.

**Session Key Barrier** The session key barrier denotes the killing of the untrusted compartment that completes session key negotiation and the subsequent spawning of a new untrusted compartment (in Linux, a process) to continue execution in the pre-authenticated stage. We now explain why this structure is necessary.

The untrusted compartment performing session key negotiation (before the session key barrier) is the only untrusted compartment in the partitioning of the cryptographic protocol implementation that processes cleartext, unauthenticated messages from the network. These messages (and exploits!) may arrive from an SKD attacker. Thus, while the untrusted compartment in the session key negotiation stage interacts with the remote peer to compute the session key, it should not have read access to the session key. In addition, any data that allows *deriving* the session key, such as a private Diffie-Hellman component (in the case of Diffie-Hellman key exchange) or a pre-master secret (in the case of RSA-based session key establishment in SSL) should be also considered sensitive. All access to privileged operations with such data should be provided via trusted compartments.

Because this compartment only processes messages in cleartext, it does not in fact need read access to the session key; only the next stage, the pre-authenticated stage, which continues execution after the channel between the two peers is MAC'ed and encrypted with the session key, needs the session key.

> Principle 1: A network-facing compartment performing session key negotiation should not have access to a session key, nor any data that allows deriving the session key.

Because the untrusted compartment performing session key negotiation may be exploited, we cannot trust the provenance of the code executing in that compartment at the end of session key negotiation, and rather than allowing that compartment to continue execution in

the pre-authenticated stage, where it would have access to the session key, we kill it (*i.e.,* kill the Linux process).

But why can't an SKD attacker exploit the untrusted compartment in the pre-authenticated stage? This compartment only processes input that is MAC'ed using the now available session key. A would-be SKD attacker cannot inject messages with a valid MAC into the channel, and so is precluded from exploiting this compartment. We assume here that the MAC computation function itself, which processes network input, can be audited and trusted not to be exploited.

Thus, both the MAC on the channel and the killing of the untrusted compartment in which session key negotiation has completed effectively erect a barrier between any SKD attacker and the session key.

> Principle 2: When enabling the MAC, a network-facing compartment performing session key negotiation should be killed, and a new one created with privilege to access the session key.

> Principle 3: After enabling the MAC, there should be no unMAC'ed messages processed by the untrusted compartment.

Note that the "original" privilege-separated OpenSSH server does in fact destroy the unprivileged compartment after user authentication, but we require this be done after key exchange. The "original" OpenSSH destroys the compartment not for SKD attack-resistance reasons, but because of a programming difficulty. In this implementation, the unprivileged compartment runs as user ID `nobody`, but must change its user ID to that of the authenticated user. Changing a process's user ID requires `root` privilege; therefore, the monitor kills the compartment and creates a new one with the required user ID.

Transitioning to the pre-authenticated stage may require transferring state from the unprivileged compartment of the session key negotiation stage to the unprivileged compartment of the pre-authenticated stage. As this state comes from a compartment that may be controlled by an SKD attacker, the pre-authentication stage should validate this state's sanity to prevent an SKD attacker from passing bad state in an attempt to compromise the pre-authenticated stage. The same problem arises when a privileged compartment accepts arguments to a privileged operation from an untrusted compartment; these arguments should also be verified to prevent compromise of the privileged compartment.

> Principle 4: Any state exported from a compartment performing session key negotiation and any untrusted arguments passed to privileged compartments should be validated.

We do not offer general techniques for verification of untrusted state and arguments. However, in our partitioning of protocol implementations, we employ pipes for inter-process communication. Although marshaling, unmarshaling, and data copies cost in performance, this mechanism provides a recipient with an RPC-like expectation of the format of the data it receives. These RPC-like semantics ease state and argument verification.

The session key barrier is enforced when an application switches permanently from communicating with cleartext messages to MAC'ed messages. Some protocols, such as SSL, however, can alternate between these two types of messages. In such cases, the transition between the two stages should be performed after the last cleartext message. However, doing so would require processing messages MAC'ed and encrypted with the session key during the session key negotiation stage, which risks creating session key oracles! We address this problem with Principle 7.

> Principle 5: A cryptographic protocol should not alternate between cleartext messages and MAC'ed messages.

**User Privilege Barrier** The user privilege barrier represents any authentication method that can be used to authenticate a peer before granting it privilege to access sensitive information owned by a particular user. This barrier prevents impersonation attacks, where an attacker exploits an application to subvert its authentication mechanism. Authentication should be performed by an unprivileged compartment that has no access to sensitive user data. The pre-authenticated stage is protected by the session key barrier, so this stage is not exposed to any SKD attacker. However, it is crucial for the integrity of the session key barrier that there be no unMAC'ed messages processed during the pre-authenticated and post-authenticated stages. Without the SKD threat, the session key is no longer sensitive information in the pre-authentication stage, and it can be accessed directly by unprivileged code. We allow the impersonator to access the session key at this point because it is his *own* key and does not correspond to any other user's session. Successful authentication transitions the application into the next stage, the post-authenticated stage.

Today's state-of-the-art privilege-reduced applications implement the user privilege barrier as we require. However, monolithic, full-privilege applications perform authentication in a privileged compartment. The privilege-separated OpenSSH server performs user authentication in an unprivileged compartment, and then the monitor creates a new compartment with the user ID and group ID of the authenticated user. The HiStar-labeled SSL web server supports only password authentication, and the unprivileged *httpd* daemon obtains ownership of the user's labels only after the user successfully authenticates with an authentication daemon.

Some protocols authenticate peers without sending confidential data, such as passwords. For example, the SSL protocol's handshake supports only public key authentication methods. Such authentication techniques can be merged with the key exchange protocol or performed in cleartext after it. Thus, the user privilege barrier can be established within the session key negotiation stage omitting the pre-authenticated stage. This optimization is encouraged, as it reduces the number of stages and compartments, and thus increases the performance of a privilege-separated application.

Authentication that requires passing sensitive data encrypted with the session key cannot be performed during the session key negotiation stage. If it were, the session key negotiation stage would require a trusted compartment to decrypt sensitive data, and that compartment would result in a session key oracle that could be used to decrypt the user's sensitive data. Moreover, other trusted compartments would be needed to process authentication-related sensitive data, because we cannot allow untrusted code to operate with confidential data.

The post-authenticated stage executes in a compartment with the authenticated user's privilege; it acts for the authenticated user and can access his data. When we transition from the pre-authenticated to post-authenticated stage, we need not kill the former, as it cannot be exploited, given the MAC'ed channel precludes SKD attacks and the authentication barrier prevents impersonation attacks. Instead, we can change the privilege of the compartment used in the pre-authenticated stage to that of the authenticated user, and continue execution with the code for the post-authentication stage.

We note that for some applications, the post-authenticated stage may require further privilege separation. For example, an application may require access to a centralized database where sensitive data belonging to many users is stored. In this case, the user-authenticated compartment should be denied direct access to the database, but a trusted compartment should export access to the database. This privilege separation, reminiscent of techniques explored in OKWS [5], prevents a user from accessing other users' sensitive data.

## 4.2 Oracle Prevention Techniques

In the previous section, we described how to implement cryptographic protocols so as to thwart SKD and impersonation attacks. Throughout the suggested implementation structure there is sensitive data accessible only by trusted compartments, which in turn export privileged operations to unprivileged compartments. As discussed in Section 3.2, in all such situations, there is a risk of granting an attacker an oracle for sensitive information.

For example, the session key negotiation stage depends on confidential session key sharing. An SKD attacker can use a trusted compartment as a decryption oracle to obtain a secret component of a session key. An impersonator may replay authentication data from another connection as an input to an authentication oracle and pass authentication as a legitimate user. Clearly, we need techniques to mitigate any oracles in these stages.

**Entangle Output Strongly with Per-Session Known-Random Input** Network protocols employ randomness generated afresh for every session to defeat authentication replay attacks, where an attacker replays messages eavesdropped from a user session to reestablish the past session and repeat a user's past requests. The server generates a random nonce incorporated into the session key (in the case of RSA key exchange) or a fresh private DH component (for DH key exchange) to make the session key different for every session. We can similarly employ this session randomness as a defense to counter oracles.

The output of a trusted compartment should not completely depend on untrusted input, so that an attacker will not be able to replay past input to the compartment and get the same deterministic result. Entangling the output of a privileged compartment with a trusted per-session random nonce solves this problem.

For example, Figure 4 demonstrates an approach to preventing a signing oracle in a privilege-separated OpenSSH server. We restrict the trusted monitor that implements signing with the private key to sign only session IDs that incorporate per-session random bits. A sequence of privileged operations performed by the trusted compartment ensures that the server's private DH component is indeed included in the session ID. This way, we entangle the output of the RSA signing compartment/operation with trusted, per-session, known-random input. Numbers within trusted compartments in Figure 4 specify the order of their invocation, and this order should be enforced by the application.

With this oracle defense mechanism, the attacker cannot mount an impersonation attack, as every signed session ID will incorporate different randomness contributed by the server, and will thus not be valid in the context of any other session. Similarly, in order to prevent deterministic session key oracles, we make sure that the compartment generating the keys includes randomness generated afresh for every session. Moreover, per-session randomness is crucial in prevention of signature verification oracles; the data for signature verification should also incorporate it.

> Principle 6: To prevent oracles, entangle output strongly with per-session, known-random input.

In RSA key exchange in the SSL/TLS protocol, there

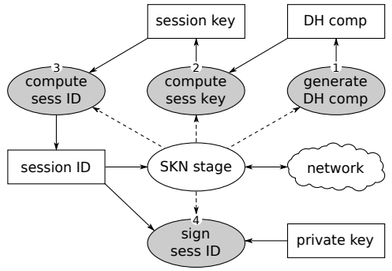



**Figure 4**: Prevention of private key oracle in OpenSSH server by entangling output with per-session known-random input.

is the potential for a deterministic session key oracle attack, where an attacker can produce a deterministic session key by supplying chosen inputs to a privileged compartment generating the key. In particular, a session key consists of two public components, per-session server and client randoms, and a pre-master secret transmitted encrypted in the server's public key [4]. When generating the session key, these components are concatenated together and hashed. The server decrypts the pre-master secret using its private key before hashing it together with the other components. If an attacker controls the server random, client random, and encrypted pre-master secret inputs to the session key generation function, he can feed data eavesdropped from a user session to the privileged compartment generating the session key and produce the key that corresponds to the eavesdropped session. We prevent deterministic session key oracles by ensuring that every server-computed session key includes a trusted server nonce produced and supplied to the compartment generating the session key by a trusted source. This way, an attacker cannot control the generated session key, as each time it incorporates a different random nonce.

**Obfuscate Untrusted Input by Hashing** The SSL protocol alternates cleartext *change cipher spec* messages with authenticated and encrypted *finished* messages [4]. A *change cipher spec* message signals that the sender is about to enable encryption and authentication on all subsequent messages. A *finished* message contains a MAC'ed and encrypted hash of all previous cleartext messages received by a peer during the handshake protocol. The *finished* message ensures that these cleartext messages were not tampered with by an attacker.

To ensure that the session key barrier is enforced, we cannot process cleartext messages in the pre-authenticated stage. Instead we should process the *finished* messages within the session key negotiation stage. However, doing so requires a trusted compartment that performs session key encryption and decryption operations on behalf of untrusted code. This trusted compartment is a session key encryption/decryption oracle which can be used to decrypt user information and validly encrypt an attacker's exploits or requests.

Our oracle mitigation technique provides the required privileged operations (encryption and decryption with a session key) and avoids a session key oracle by obfuscating input data through hashing. As the *finished* message is an encrypted hash, a trusted compartment can be structured in the following way: it obtains data from an untrusted compartment, hashes the data, and then encrypts the resulting hash. A privileged operation that hashes data and then encrypts is not useful for an attacker, as the attacker's requests and exploits for the pre-authenticated and post-authenticated stages will be viewed as hashes.

As for the decryption oracle, we do not return the cleartext *finished* message to untrusted code. Instead, our trusted compartment takes the verification data from an untrusted compartment and performs verification of the *finished* message itself. The result of this verification is returned to the untrusted compartment. However, this mechanism allows dictionary attacks, where an attacker can guess the cleartext message by supplying the verification data. Again, obfuscating the untrusted validation data by hashing before comparing it with the cleartext *finished* message solves this problem. This approach fits the protocol because the *finished* message happens to be a hash of all previous handshake messages. If an attacker attempts to guess the cleartext requests, his guess will be hashed first, then compared with the original message.

The hashing that we apply to prevent both oracles already is present in the SSL handshake. But the handshake and our oracle mitigation technique use it for different reasons. The handshake requires the compression and collision-resistance of a hash function, but our technique employs the hash function because of its non-invertibility. Happily for us, the hash function provides all the mentioned properties and does double duty.

> Principle 7: To prevent oracles, obfuscate untrusted input by hashing.

**Last Resort: More Trusted Code** The previous oracle mitigation techniques require the availability of a random nonce or a hash function. However, for those cases in which a cryptographic protocol does not specify these functions at a point in the protocol where there is the risk of an oracle, we offer a last resort technique.

For an oracle to exist, a result of a privileged operation must return to an unprivileged compartment. It is possible to avoid the oracle by making the output privileged and restricting access to it in the unprivileged code. Although this technique helps, it is not efficient, as a new trusted compartment is required to process the result, and you may need to process the result of the new compartment in the same way. Our last resort technique may lead to a chain of trusted compartments, which increases the trusted code base and requires more auditing work. Moreover, to terminate this chain, there must be a suitable condition for applying one of the previous oracle mitigation techniques, or the last trusted compartment in the chain must not produce any output.

> Principle 8: To prevent oracles, as a last resort, add more trusted code.

### 4.3 Degrees of Sensitivity

Cryptographic protocols often operate on sensitive data of more than one class. As an example, one frequently occurring class of sensitive data is that which must be kept secret to ensure secrecy and integrity of data transferred within a *single* session, *e.g.,* the pre-master secret in RSA key exchange, the private DH component in DH key exchange, the session key, the per-session ephemeral RSA private key, *&c.* Disclosure of such sensitive data results in violation of the secrecy and/or integrity of sensitive data within a single session. Yet there is often another class of even more sensitive data that must remain secret in order to preserve the secrecy of user data in *many* sessions. This class includes a server's private key, users' private keys, and passwords that are reused on many servers. The secrecy of such data is vital because an attacker can use it to gain access to user data in multiple sessions by impersonating the server, or by using users' passwords to access many servers.

In a simple scenario like this one involving two classes of sensitive data—that which is critical to one session's secrecy *vs.* that which is critical to ensuring many sessions' secrecy—mixing sensitive data of both classes and code to manipulate data of both classes in the same compartment incurs warrantless risk. To see why, let's deviate from our threat model and assume that an attacker can compromise trusted compartments. Now any vulnerability in code that manipulates sensitive data pertaining to one session's secrecy can disclose sensitive data that could compromise secrecy of all sessions. Creating distinct compartments for data of differing degrees of sensitivity (and the code that manipulates it) mitigates this risk. Similarly, to prevent disclosure of one user's data to another, separate compartments should manage sensitive session-related key data for each user.

> Principle 9: A privilege-separated application should manage a session with two separate privileged compartments—one to operate with data related to secrecy of the current session, and one to manage data that preserves secrecy of many sessions.

Isolating code and data in distinct compartments according to their sensitivity often reduces trusted code base size; the quantity of code with privilege with respect to one piece of data decreases.

## 5 Hardened SSH Protocol Implementation

We now demonstrate these principles for preventing SKD and oracle attacks by finely privilege-separating the implementations of the client and server sides of the SSH protocol.

Recent privilege separation and DIFC work focuses on server applications, as they accept connections and can thus be attacked at will. But the rise of web browser exploits demonstrates that client code is equally at risk. An attacker can set up a public service and provide access to it via SSH. By exploiting vulnerabilities in the SSH client implementation, the attacker can obtain users' private keys, used to authenticate them to other legitimate SSH servers. These keys allow the attacker to obtain or tamper with the user's sensitive information stored at these other SSH servers. Moreover, as the SKD attack is equally valid on both sides, server and client, protection against it is equally needed on the two sides.

Throughout this paper, the baseline OpenSSH server design we refer to is that of Provos *et al.* [9]. While this OpenSSH server implements privilege separation, it allows unprivileged code access to the session key (contravening Principles 1 and 2) and to sign a session ID provided by unprivileged code (contravening Principle 6), and thus is vulnerable to SKD and oracle attacks. We show how to partition the server more finely to prevent these attacks. But first, we focus on the OpenSSH client, which to date has only existed in monolithic form, and is thus also vulnerable to both attacks.

### 5.1 Hardened OpenSSH Client

The OpenSSH client runs under the invoking user's user and group IDs. Because changing the user ID to `nobody` and invoking the `chroot` system call require `root` privilege, they cannot be used here. Instead, we limit the privilege of the trusted and untrusted compartments of the OpenSSH client with SELinux policies [7], and the SELinux type enforcement mechanism in particular. SELinux policies allow us to restrict untrusted processes from issuing unwanted system calls such as `ptrace`, `open`, `connect`, *&c.*[3] Our prototype supports only password and public key authentication, and does not yet implement advanced SSH functionality (tunneling, X11 forwarding, or support for authentication agents).

Our hardened OpenSSH client starts in the *ssh_t* domain, defined as a standard policy in the SELinux package for the original monolithic SSH client. This policy provides the union of all privileges required by all code in the SSH client; *i.e.,* an application in the *ssh_t* domain may open SSH configuration files, access files in the `/tmp` directory, connect to a server using a network

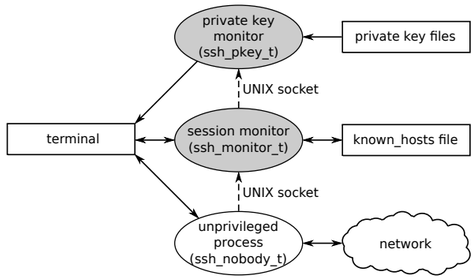



**Figure 5**: Architecture of privilege-separated OpenSSH client. Shaded ovals denote privileged compartments. Unshaded ovals denote unprivileged compartments. The last line in each oval denotes the SELinux policy enforced.

| Session monitor |
|---|
| 1) **DH_priv_key** = gen_DH_priv_key() |
| 2) DH_pub_key = comp_DH_pub_key(**DH_priv_key**) |
| 3) **sess_key** = comp_sess_key(**DH_priv_key**, srvr_DH_pub_key) |
| 4) **sess_ID**$^i$ = comp_sess_ID(**sess_key**, clnt_version, srvr_version, clnt_kexinit, srvr_kexinit, ...) |
| 5) **sym_keys** = derive_sym_keys(**sess_ID**$^i$, **sess_key**) |
| 6) **srvr_pub_key**$^i$ = verify_srvr_pub_key(srvr_pub_key, **known_hosts_file**) |
| 7) verify_ sig(**sess_ID**$^i$, **srvr_pub_key**$^i$, sig) |
| Private key monitor |
| 1) sig = priv_key_sign(**priv_key**, **sess_ID**$^i$, user_name, service, auth_mode, ...) |

**Figure 6**: Privileged operations performed by the two client monitors. Sensitive data appear in bold, and are accessible only by the monitor compartment in which they appear. Untrusted parameters provided by unprivileged compartments are not in bold. $\mathbf{x}^i$ denotes that sensitive data **x** is exported to an unprivileged compartment read-only.

socket, create a pseudo-terminal device, *&c.* We use this domain to initialize the client application and connect to the requested SSH server. At this point, the client has not yet processed any data from the server. Before exchanging any SSH protocol messages, the client creates two new processes (compartments): a privileged *session monitor* that performs privileged operations on sensitive data that can compromise only a single SSH session, and a *private key monitor* that performs authentication operations with the client's private keys. This ensemble of three compartments (represented by ovals) appears in Figure 5. The use of two distinct monitors is motivated by Principle 9.

The session monitor runs in the *ssh_monitor_t* domain, a domain we have defined that confines the process to access only the `known_hosts` file; to read/write UNIX sockets for communicating with the private key monitor and an unprivileged process running untrusted code (described below); and to read/write a terminal device. The session monitor cannot create or access any files apart from `known_hosts`, nor may it create new sockets. The private key monitor runs in the *ssh_pkey_t* domain, a domain we have defined with a similarly tight policy, allowing it only to read the user's private key(s), with no access to other files, nor privilege to create any sockets. The private key monitor shares a UNIX socket with the session monitor and only accepts requests from the latter. After creating these two monitor processes, the original SSH client process drops privilege to the *ssh_nobody_t* domain. Untrusted code runs in this unprivileged process and domain during the rest of the SSH client's execution. The *ssh_nobody_t* domain allows the unprivileged process to communicate with the session monitor and remote server via previously opened sockets, but prevents it from opening any new ones. The *ssh_nobody_t* domain further denies all access to the file system, allowing the unprivileged process access to the terminal device only.

The session monitor compartment isolates all sensitive data that can be used to compromise the current remote login session, and performs all privileged operations with these data, enumerated in Figure 6, that are essential for key exchange and prevention of a private-key oracle. When a privileged operation takes non-sensitive data as input, the non-sensitive input is supplied by the unprivileged compartment. *Symmetric keys* (sym_keys) are the keys derived from the session key for the MAC and encryption/decryption. The session monitor enforces the order in which an untrusted compartment may invoke its privileged operations.

The private key monitor isolates the client's private key and performs signing operations with the key. Only the session monitor may invoke these signing operations in the private key monitor (over a UNIX-domain socket), and it provides the session ID to be signed as an argument. We give a more detailed explanation of the private key signing operation at the end of this section.

**Session Key Negotiation Stage** We now consider the first stage of the hardened OpenSSH client, the session key negotiation (SKN) stage, designed to thwart SKD attacks (described in Section 3.1). In the SKN stage, an unprivileged compartment—with the help of the session monitor—performs Diffie-Hellman key exchange to negotiate a session key and authenticate the server. In accordance with Principle 1, we restrict the SKN stage to run in an unprivileged compartment that cannot access sensitive data—not the DH private key, nor the session key, nor the symmetric keys (as shown in Figure 6). Keeping the session key secret (and thus thwarting an SKD attack) requires in turn keeping this data secret.

We must also prevent a *verification oracle attack* against the client at this point in the handshake. Suppose the attacker wants to impersonate a server to the client, and can trick the client into connecting to a server he controls, instead of to the bona fide server intended by the client. Suppose further that the attacker exploits the client. To authenticate the server, the client must verify the server's public key against the list of trusted public keys in the `known_hosts` file, and then validate the server's signature on the session ID. Once the attacker exploits the client, if the exploited compartment of the client implementation allows invocation of signature verification operation with the session ID or server's public key provided by this compartment then the attacker may be able to force signature verification to succeed, and thus spoof the bona fide server to the client. To see why, note the arguments to the signature verification routine verify_sig() in the session monitor in Figure 6. If the attacker controls the values of the signature argument and *either* the session ID argument *or* the server public key argument, he can provide inputs that will cause the signature to verify. That is, he can either sign a benign sess_ID with his *own* private key and supply his *own* corresponding srvr_pub_key, or supply a bogus sess_ID signed by the bona fide server (readily obtained from the attacker's own connection to the bona fide server), along with the bona fide server's true srvr_pub_key.

To prevent this verification oracle, we must not allow an unprivileged compartment (at risk for exploit) to provide either srvr_pub_key or sess_ID to verify_sig(). We thus perform signature verification in the session monitor, and isolate sess_ID and srvr_pub_key within the monitor. In actuality, the untrusted compartment provides srvr_pub_key to the session monitor, but the session monitor validates it against the contents of the `known_hosts` file before verifying the signature. Note that sess_ID is entangled with trusted random bits generated by the client every new session, originating from the client's DH_priv_key via comp_sess_key() and comp_sess_ID(). This construction, specified by the OpenSSH protocol, implicitly applies Principle 6, which further prevents an attacker from forcing sess_ID to match that from a past eavesdropped session.

We now turn our attention to the next steps taken by the client. In the OpenSSH protocol, session key negotiation and server authentication, which establishes the user privilege barrier, are intertwined. Therefore, our partitioning of OpenSSH needs no distinct pre-authenticated stage, and the SKN stage proceeds immediately to the post-authenticated stage.

**Post-authenticated Stage** After computing symmetric keys and authenticating the server, the client kills the untrusted compartment from the SKN stage and creates a new untrusted compartment, also confined to the *ssh_nobody_t* domain, to execute operations in the post-authenticated stage. This new compartment is granted access to the session's symmetric keys so that it can perform encryption and decryption operations. It may invoke privileged operations in the session monitor, and the session monitor can invoke privileged operations on the client's private keys by the private key monitor. To do so, the private key monitor executes with the privilege to read private key files.

In the post-authenticated stage, the server authenticates the client. Our prototype supports password and public key authentication. Password authentication does not require any further partitioning of the client to protect against a malicious server, as the SSH protocol requires that the client sends the password to the server. However, we can apply fine-grained privilege separation to deny the server access to the client's private key(s). There is no need for the untrusted compartment to have direct access to the keys, and if it does, a malicious server that the user logs in may exploit the client and obtain its private keys, and thus obtain sensitive information from other SSH servers where the user authenticates himself using the same private keys. Therefore, we isolate the client private keys from the post-authentication stage's untrusted compartment by placing them in a privileged private key monitor. To prevent a private key signing oracle in the client, we do not allow the untrusted compartment to directly invoke signing data of its own choice using the private key. The untrusted compartment passes untrusted input (user name, service name, authentication mode, *&c.*) via the session key monitor. Note that we rely on session key monitor to supply the trusted session ID computed earlier in the key exchange protocol to the private key monitor as shown in Figure 6. Recall that the session ID has been entangled with trusted random bits generated by the client for the current session. Thus, the signature produced by the private key monitor will not be valid in any session but the current one, and a private key oracle has been disseminated.

To support session key rekeying, the unprivileged process is permitted to invoke privileged rekeying operations implemented by the session monitor.

## 5.2 Hardened OpenSSH Server

In accordance with Principle 9, we extend the baseline privilege-separated OpenSSH server with an extra session monitor process that handles sensitive data related to a single user's session while preventing an SKD attack and both private key signing and signature verification oracles, as shown in Figure 7. The private key monitor is the original monitor process from the baseline privileged-separated OpenSSH server, which performs operations that require `root` privilege.

The session monitor, the unprivileged SKN process, and the unprivileged process of the pre-authentication stage all run in a `chroot`ed environment with an unused

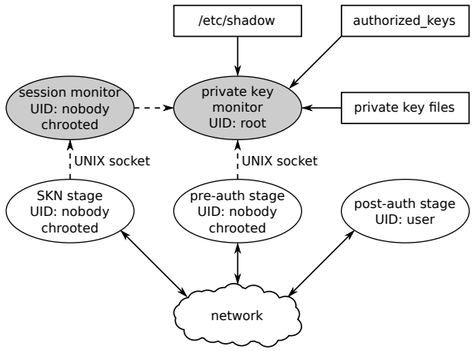



**Figure 7**: Architecture of hardened OpenSSH server.

UID, under a restrictive SELinux policy that allows only the system calls implied in Figure 7, and prohibits all others, including dangerous ones such as `ptrace` and `connect`. The process for the post-authenticated stage runs with the UID of the authenticated user and is not restricted with any SELinux policy, as with the baseline OpenSSH server.

**Session Key Negotiation Stage** The session monitor implements the privileged operations required for the SKN stage, and we ensure that the pre-authenticated stage does not start unless the unprivileged compartment of the SKN stage terminates (in accordance with Principle 2). Because the Diffie-Hellman key exchange protocol is symmetric between the server and client, we implement operations 1–5 from Figure 6 in the server's session monitor just as in the client's. The SKD attack is an equally serious threat for client and server; as both parties share the same session key, an SKD attacker can compromise either party's code to disclose it.

During the SKN stage, the server authenticates itself to the client by signing a session ID. The monitor in the baseline privilege-separated OpenSSH server signs any data supplied by the untrusted compartment, thus allowing an oracle attack. A man-in-the-middle attacker can interpose himself between a client and a bona fide server and employ a signing oracle on the server to impersonate the server by producing valid signatures on session IDs corresponding to the attacker's session with the client. We prevent such attacks by constraining the private key monitor to sign only data provided by the trusted session monitor—specifically, the current session ID entangled with trusted random bits provided by the server, as shown in Figure 4, as suggested by Principle 6. The server's session monitor produces this sess_ID in operation 4 in Figure 6, just as the client's does. This signed sess_ID cannot be used to impersonate the server as it is only valid within the current session. To perform the signing operation, the session monitor calls into the privileged private key monitor and supplies the required trusted sess_ID to sign.

**Pre-authenticated and Post-authenticated Stages** The baseline privilege-separated OpenSSH server separates the pre-authenticated and post-authenticated stages. It performs user authentication operations such as password verification and signature validation (in public key authentication) in the monitor. However, this architecture allows an SKD attacker to compromise the password during password authentication, as it is encrypted with the session key obtainable by the attacker. During public key authentication, the untrusted compartment supplies the data used for user signature verification, again allowing oracle attacks against user authentication. The monitor validates the signature against the session ID supplied earlier when the untrusted compartment requested the server's signature on this session ID. Thus the untrusted compartment can control the session ID used in public key authentication of the user. In order for an attacker to impersonate the client, she must provide some session ID signed by the client for the server's verification operation. It is unlikely that the attacker can force a user to sign arbitrary data with his private key. However, an SKD attacker can compromise the user's session and log its session ID and signature pair. She can then replay these data to the server's signature verification compartment. Because the server's signature verification routine does not check whether the provided session ID is valid within the current session, the verification routine will report that the client has authenticated successfully. In this way, the attacker successfully impersonates the user.

In our implementation, we fix this problem by making sure that the session ID used for signature verification is produced by the session monitor, as done in operation 4 in Figure 6, and entangled with trusted random bits provided by the server. Our SKN stage also ensures the secrecy of user passwords by thwarting SKD attacks.

**Discussion: Trusted Code Base** Figure 8 compares the trusted code bases of Provos *et al.*'s baseline privilege-separated OpenSSH server and our hardened OpenSSH server. The latter implements two monitors, in accordance with Principle 9, and as described in Figure 7: one private key monitor that implements code required for user authentication and accessing the server's private key, and one session key monitor that contains the privileged code for processing the sensitive state for a user's session. Consider operations 1–5 in Figure 6, which are essential to protection against SKD and oracle attacks. In our partitioning, the session monitor implements these five operations, while in baseline OpenSSH, the untrusted compartment implements them.



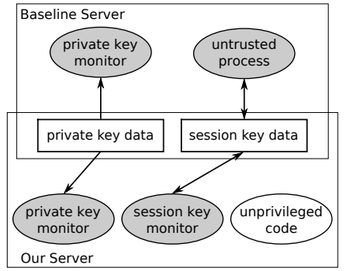



**Figure 8**: Relationship between privileged (shaded) and unprivileged (unshaded) code in baseline and hardened OpenSSH server implementations.

At first glance, one might remark that our partitioning therefore incorporates *more* privileged code than baseline OpenSSH. But that assessment is flawed. Rather, the sensitive state pertaining to a user's session was incorrectly deemed non-sensitive data in baseline OpenSSH. Hence, we show baseline OpenSSH's untrusted process as shaded—notation for privileged—because it is *already* (albeit inappropriately) privileged to manipulate sensitive per-session data. Following the partitioning principles we have offered leads to the correct treatment of this data as sensitive, the creation of a new privileged compartment that can exclusively manipulate this data (the session monitor), and the *reduction* of privilege for all remaining code from baseline OpenSSH's untrusted process (denoted in the figure as "unprivileged code")!

## 6 Hardened OpenSSL Library

Toward demonstrating the generality of the partitioning principles presented in Section 4, we have also applied them to the SSLv3 and TLSv1 cryptographic protocol implementations in the OpenSSL library. As partitioning in accordance with these principles requires a fair amount of programmer effort, we found the OpenSSL library a particularly attractive target; hardening the library allows amortizing one partitioning effort over a broad range of security-conscious applications. The resulting hardened OpenSSL library is a drop-in replacement that renders any SSL/TLS application linked against it immune to SKD and oracle attacks. We note, however, that changing the library alone cannot ensure that the application *atop* the library itself handles sensitive data securely. For example, the Apache web server reuses worker processes across requests submitted by different users. If an attacker exploits a worker process, he may be able to obtain sensitive data belonging to the next user whose request is handled by that process.

We finely partition both the client and server sides of OpenSSL. Our implementation supports RSA, ephemeral RSA, Diffie-Hellman, and ephemeral Diffie-Hellman key exchange, client and server authentication, and session caching. The OpenSSL partitioning is in fact similar in structure to that of SSH, as these protocols protect against similar threat models. When an application invokes SSL_accept (on the server) or SSL_connect (on the client), we instantiate private key monitor, session key monitor, and unprivileged SKN compartments. Our implementation scrubs the server's private key from the session key monitor process and the unprivileged SKN compartment before reading any input from the network. Within the SKN stage, we apply the same principles and mechanisms as we did to OpenSSH to prevent SKD and oracle attacks. As SSL/TLS supports only public key authentication, its partitioning omits the pre-authentication stage. We apply simple SELinux policies (whose details we elide in the interest of brevity) to limit the privilege of the untrusted SKN compartment and the session monitor in applications that do not run as `root`. When the SKN stage completes, the unprivileged compartment and session monitor are terminated, and execution continues in the application's fully privileged compartment. The private key monitor preserves the privileges of the application before entering the SSL_accept and SSL_connect library calls. Therefore, this compartment continues execution of the application's code and can use the symmetric key computed during the SSL handshake to perform MAC and encryption/decryption operations on the established SSL/TLS session.

We have tested this hardened OpenSSL library with a number of client-side and server-side applications, including the server and client sides of stunnel, the mutt and mailx mail agents (for IMAP and POP3 over SSL/TLS), the dovecot IMAP and POP3 server, the client and server sides of the sendmail mail transfer agent (for SMTP over SSL/TLS), and the Apache HTTPS server (versions 1.3.19 and 2.2.14).

Converting most of these applications was straightforward; it merely required replacing the OpenSSL library and making a one-line change to the application's SELinux policy, without any application code modifications. Apache, however, required code modifications—not to protect against SKD and oracle attacks, which the partitioned OpenSSL library defends against, but to protect sensitive data *after* the SSL handshake completes. As noted above, Apache reuses worker processes to serve successive users' requests. We modified Apache to enforce *inter-user isolation:* to ensure that an attacker's exploit of a worker cannot disclose the sensitive data of the next user to connect to the same worker. We compare two implementations of this isolation. The first is a naive one in which Apache kills a worker after it serves one request and `fork`s another to replace it. As the overhead of `fork` is significant, we compare against an optimized implementation based on *checkpoint-restore*, as

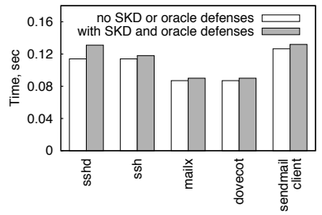



**Figure 9**: Latency of operations in OpenSSH 5.2p1 client/server, mailx 12.4, dovecot 1.2.10, and sendmail client 8.14.4 using baseline and hardened OpenSSL 0.8.9k library. Run on Dell desktop with 1.86 GHz Intel Core 2 6300 CPU and 1 GB RAM running Linux 2.6.30.

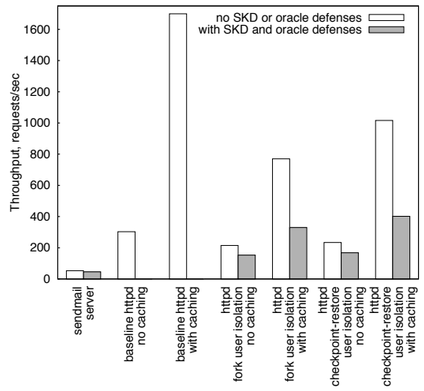



**Figure 10**: Throughput of sendmail server 8.14.4 and indicated combination of Apache web server (httpd) 2.2.14 with OpenSSL 0.8.9k library. Run on Sun X4100 server with 2.2 GHz AMD Opteron 248 CPU and 2 GB RAM under Linux 2.6.32.

proposed by Bittau [1]. In this approach, Apache takes a snapshot of each new worker process's pristine memory image before it serves any requests, and after each request, a trusted monitor process restores the worker's memory image to this pristine state.

With or without this unrelated application-level change, Apache 1.3.19 and 2.2.14 run with the hardened OpenSSL library as a drop-in replacement for the stock OpenSSL library.

## 7 Evaluation

We now consider the cost of defending against SKD and oracle attacks in cryptographic protocol implementations. As the principles given in Section 4 demand additional isolation between code and data, and thus additional processes, performance is a concern: both process creation and context switches incur overhead. To explore the extent of these overheads, we compare the performance of the baseline OpenSSH and OpenSSL-enabled applications with that of the implementations hardened in accordance with the principles we have propounded. We consider in turn the end-to-end metrics of operation latency (important to users) and server-side throughput (important to server operators).

Figure 9 compares operation latencies for a range of applications. Each application is either client-side or server-side; in each case, the complementary remote peer runs the *baseline* cryptographic protocol implementation. All connections are made over the loopback interface to a locally running server. For OpenSSH, we report the latency of logging into an SSH server using public key authentication and running the `exit` command. The remaining applications use the OpenSSL library. For the mailx email client and dovecot IMAP server, we measure the time required for the client to connect over SSL/TLS, check for new mail, and exit. For the sendmail client, we measure the time required to connect and send a one-line email to a sendmail server over SSL/TLS. For these applications, the latency a user perceives does not increase significantly between the baseline and hardened cryptographic protocol implementations.

In Figure 10, we consider the throughput achieved by an SSL/TLS-enabled sendmail server and HTTPS server, both based on the OpenSSL library. For the sendmail server, we submit emails over SSL/TLS from multiple clients and report the maximum load the server can sustain in requests (emails) per second. Introducing oracle and SKD defenses into the OpenSSL library negligbly affects the sendmail server's throughput.

To determine the maximum load the Apache (httpd) web server can sustain, we increase the number of clients requesting a small static page over HTTPS until the number of requests served per second reaches a maximum. Clients make new SSL/TLS connections for each request. As noted in Section 6, apart from adding defenses against SKD and oracle attacks, we further modified the baseline Apache implementation to isolate users who successively connect to the same worker from one another. To distinguish the cost of inter-user isolation from that of defending against SKD and oracle attacks, we measure the throughput of several Apache implementations: baseline Apache, in which workers are reused across requests, so users are not mutually isolated; a hardened Apache with inter-user isolation implemented with one `fork` per request, *without* oracle or SKD attack defenses; and a hardened Apache with inter-user isolation implemented with three `fork`s per request, *with* oracle and SKD attack defenses. To explore the role of isolation primitives in performance, we also implemented versions of hardened Apache that use optimized checkpoint-restore primitives [1] rather than `fork`. We further consider Apache's performance in two extremes of operation: when no SSL sessions are cached and when all sessions are cached. We configure HTTPS clients to use RSA key exchange when establishing an SSL/TLS session because this protocol is less computationally intensive for the server than ephemeral Diffie-Hellman key exchange, and thus better exposes the overhead of hardening.

Returning to Figure 10, let us first consider the workload in which no SSL/TLS sessions are cached, running on the hardened versions of Apache implemented using checkpoint-restore. End-to-end, the version of Apache providing both inter-user isolation and defenses from oracle and SKD attacks achieves more than half (55%) the throughput of baseline Apache, which provides none of these security benefits. The overhead of these security mechanisms is masked in part by the computational costs of the cryptographic operations required to establish a new SSL/TLS session. We note that this "fully" hardened version of Apache achieves over 70% the throughput of one that provides inter-user isolation with checkpoint-restore but omits oracle and SKD attack defenses—so for this workload using these isolation primitives, oracle and SKD attack defenses incur only moderate overhead.

In the workload in which no SSL/TLS sessions are cached, there are no public-key cryptographic operations, so the overheads of inter-user isolation and oracle and SKD attack defenses are more exposed. Focusing on the implementations built on checkpoint-restore, Apache with inter-user isolation (but without oracle/SKD defenses) achieves 60% of the throughput of baseline Apache; this reduction is the cost of inter-user isolation. Adding oracle and SKD defenses to the inter-user-isolated implementation further reduces throughput by 60%; that is the incremental cost of oracle and SKD defenses on this challenging workload. End-to-end, this last version of Apache, which incorporates all defenses and inter-user isolation, achieves only about one quarter of the throughput of baseline Apache (which lacks any of these security enhancements). We stress that while this throughput reduction is significant, it represents atypically worst-case behavior: all sessions cached (never the case) and static content. On servers that distribute dynamically generated content, the overhead of protecting users' sensitive data will be amortized over far more application computation.

The original applications based on the OpenSSL library used single-process, monolithic designs. Hardening against SKD and oracle attacks requires three processes per SSL/TLS session: a private key monitor, a session monitor, and an unprivileged compartment for the SKN stage. Similarly, the hardened OpenSSH server and client use four processes per SSH session *vs.* the two employed by the baseline privilege-separated OpenSSH server. Apart from the process creation and page fault costs associated with `fork` and the memory copy costs associated with checkpoint-restore, anti-SKD and anti-oracle hardening incur overhead for additional context switches and the marshaling and unmarshaling of arguments and return values between compartments connected by pipes.

Again for the uncached workload, consider the throughput achieved by the full checkpoint-restore version of Apache (all defenses) *vs.* that achieved by one with the same full set of defenses, but implemented naively with `fork`. Checkpoint-restore offers a 20% throughput improvement over `fork`. While the end-to-end cost of inter-user isolation and oracle and SKD defenses is significant, the design of the underlying primitives used to implement compartments, though beyond the scope of this paper, appears to play a significant role in determining end-to-end performance.

## 8 Related Work

Provos *et al.* describe privilege separation, which denies enhanced system privileges to unauthorized attackers who exploit an application [9]. They reduce privilege in the OpenSSH server by partitioning it into an untrusted process and a privileged monitor. Our work tackles the different goal of preventing disclosure of users' sensitive data in cryptographic protocol implementations. This goal incorporates preventing privilege escalation. We extend the partitioning of the privilege-separated OpenSSH server to comply with this goal.

OKWS is a toolkit for building secure Web services [5]. It employs similar privilege enforcement mechanisms as privilege-separated OpenSSH—processes, the `nobody` user ID, and the `chroot` system call—to isolate distrusted Web services from the system they are running on and each other. Our complementary goal has been to protect sensitive data by hardening cryptographic protocol implementations against exploit.

HiStar [14] enforces privileges on compartments with labels and DIFC. DStar [15] extends this approach to a distributed environment without fully trusted machines. Zeldovich *et al.* partition an SSL server to mitigate the effect of a compromise of any single compartment and prevent disclosure of user data. However, as we have described, it is possible to disclose users' sensitive data from the SSL server using SKD and oracle attacks. The insufficient partitioning of the SSL protocol allows these attacks. Our work is complementary to work on DIFC systems: they are privilege-enforcement mechanisms, while we provide guidance on how to structure code for cryptographic protocols.

We first discovered an instance of the attack we have generalized in this paper as the SKD attack during prior work with colleagues on Wedge [2], a set of primitives and tools for fine-grained partitioning of applications on Linux. While we presented an ad hoc defense for one narrow instance of the attack in that work, we offered no general characterization of it nor solution to it. By contrast, in this paper, we offer design principles that defeat

the SKD and oracle attacks and that we believe are general enough to apply to many cryptographic protocols.

The partitioning principles and attack mitigation techniques we have offered might also find fruitful use in capability-based systems such as KeyKOS [3] and EROS [11]. While capabilities provide convenient means to restrict privileges, programmers need guidance in how to *apply* them to protect sensitive data.

## 9 Conclusion and Future Work

We have described two practical exploit-based attacks on cryptographic protocol implementations, the session key disclosure (SKD) attack and oracle attack, that can disclose users' sensitive data, even in state-of-the-art, reduced-privilege applications such as the OpenSSH server and HiStar-labeled SSL web server. Privilege separation and DIFC will not secure the user's sensitive data against these attacks unless an application has been specifically structured to thwart them.

The principles we have offered guide programmers in partitioning cryptographic protocol implementations to defend against SKD and oracle attacks. In essence, following these principles reduces the trusted code base of an application by correctly treating session key material and oracle-prone functions as sensitive, and limiting privilege accordingly.

To demonstrate that these principles are practical, we newly partitioned an OpenSSH client and extended the partitioning of a privilege-separated OpenSSH server. Further experience with the OpenSSL library suggests they may generalize to other cryptographic protocols; they are broadly targeted at protocols that negotiate session keys and perform common cryptographic operations. While we hope these principles will serve as a useful guide where there was none, we note that their application requires careful programmer effort. Still, our experience with OpenSSL shows that hardening a library once brings robustness against these attacks to the several applications that reuse that library.

The latency cost of defending against SKD and oracle attacks is well within user tolerances for all applications we measured. Defending against SKD and oracle attacks does exact a cost in throughput on a busy SSL-enabled Apache server, however, reducing the uncached SSL/TLS session handshake rate of a server that isolates users by just under 30%, and the cached rate by 60%. While that cost is significant, as our comparison of `fork` and checkpoint-restore demonstrates, it depends heavily on the performance of underlying isolation primitives—a topic we believe merits further investigation.

Finally, while we have relied upon manual study of the SSH and SSL/TLS protocols and their implementations to discover the attacks we have presented, we intend to explore tools that use static and dynamic analysis to ease discovery of such vulnerabilities in cryptographic protocol implementations.


## Acknowledgements

This research was supported in part by a Royal Society-Wolfson Research Merit Award and by gifts from Intel Corporation and Research in Motion Limited. We thank Andrea Bittau, our shepherd Mohammad Mannan, and the anonymous reviewers for comments that improved the paper. We further thank Andrea Bittau for sharing code for his checkpoint-restore server performance optimizations.


## References


[1] A. Bittau. *Toward Least-Privilege Isolation for Software*. PhD thesis, University College London, UK, 2009. http://eprints.ucl.ac.uk/18902/1/18902.pdf.

[2] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *NSDI*, 2008.

[3] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, 1992.

[4] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, January 1999.

[5] M. Krohn. Building secure high-performance web services with okws. In *USENIX*, 2004.

[6] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[7] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX (Freenix Track)*, 2001.

[8] N. Provos. Improving host security with system call policies. In *USENIX Security Symposium*, pages 18–18, 2003.

[9] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security*, 2003.

[10] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[11] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: a fast capability system. In *SOSP*, 1999.

[12] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the asbestos operating system. *ACM TOCS*, 25(4):11, 2007.

[13] T. Ylonen and C. Lonvick. The secure shell (SSH) protocol architecture. RFC 4251, January 2006.

[14] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[15] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *NSDI*, 2008.


## Notes

[1] While we did not implement these two attacks, we present analysis of the protocols and implementations demonstrating they are possible.

[2] While space limits us to illustrating these attacks and defense principles in the context of SSH and SSL/TLS, we have found they apply equally to IPSec, CRAM-MD5, and other secure protocols.

[3] Alternatives to SELinux include limiting a process's privileges with Systrace [8], `ptrace`, and `chroot` (though the latter requires making a client application `setuid root`).



# PrETP: Privacy-Preserving Electronic Toll Pricing


*Josep Balasch, Alfredo Rial, Carmela Troncoso,*
*Bart Preneel, Ingrid Verbauwhede*
*IBBT-K.U.Leuven, ESAT/COSIC,*
*Kasteelpark Arenberg 10,*
*B-3001 Leuven-Heverlee, Belgium.*
`firstname.lastname@esat.kuleuven.be`

*Christophe Geuens*
*K.U.Leuven, ICRI,*
*Sint-Michielsstraat 6,*
*B-3000 Leuven, Belgium.*
`christophe.geuens@law.kuleuven.be`



## Abstract

Current Electronic Toll Pricing (ETP) implementations rely on on-board units sending fine-grained location data to the service provider. We present PrETP, a privacy-preserving ETP system in which on-board units can prove that they use genuine data and perform correct operations while disclosing the minimum amount of location data. PrETP employs a cryptographic protocol, Optimistic Payment, which we define in the ideal-world/real-world paradigm, construct, and prove secure under standard assumptions. We provide an efficient implementation of this construction and build an on-board unit on an embedded microcontroller which is, to the best of our knowledge, the first self-contained prototype that supports remote auditing. We thoroughly analyze our system from a security, legal and performance perspective and demonstrate that PrETP is suitable for low-cost commercial applications.


## 1 Introduction

Vehicular location-based technologies [36, 42] are viewed by governments as a perfect tool to support applications such as electronic toll collection, automated law enforcement, or collection of traffic statistics. In October 2009, the European Commission announced that the current flat road tax systems existing in the Member States will be substituted by an European Electronic Toll Service (EETS) [13, 20]. In the United States, there are also ongoing initiatives to introduce *Electronic Toll Pricing* (ETP), as for instance the Regional High Occupancy Toll Network of the California Metropolitan Transportation Commission [1].

ETP allows road taxes to be calculated depending on parameters such as the distance covered by a driver, the kind of road used, or the time of usage. This is beneficial both for citizens and governments. The former pay only for their actual road use, while the latter can improve road mobility by applying "congestion pricing". This strategy assigns prices to roads depending on their traffic density such that driving in congested roads implies a higher cost. This in turn will encourage users to change their route (or even avoid using their vehicles) thus reducing congestion. Moreover, ETP has also environmental benefits as it discourages driving hence reduces pollution.

ETP architectures proposed so far [1, 13, 20] require that vehicles are equipped with an on-board unit necessary for collecting location data. At the end of each tax period, the fee corresponding to those data is computed either remotely [36, 42] or locally [44], and relayed to the service provider. In both cases the service provider needs to be convinced that the fees correspond to the actual road usage of the driver, and that they have been correctly calculated. The verification is straightforward in implementations in which all the location data is sent to the service provider, but this constitutes an inherent threat to users' privacy.

We propose PrETP, a privacy-preserving ETP system in which, without making impractical assumptions, on-board units i) compute the fee locally, and ii) prove to the service provider that they carry out correct computations while revealing the minimum amount of location data. PrETP employs a cryptographic protocol, Optimistic Payment (OP), in which on-board units send along with the final fee commitments to the locations and prices used in the fee computation. These commitments do not reveal information on the locations or prices to the service provider. Moreover, they ensure that drivers cannot claim that they were at any other position, nor used different prices, from the ones used to create the commitments. In order to check the veracity of the committed values, we rely on the service provider having access to a proof (e.g., a photograph taken by a road-side radar or a toll gate) that a car was at a specific point at a particular time, as previously suggested in [17, 39]. Upon being challenged with this proof, the on-board unit must respond with some information proving that the location

point where it was spotted was correctly used in the calculation of the final fee. To this end, it opens the commitment containing this location, thus revealing *only* the location data and the price at the instant specified in the proof. This information suffices for the provider to verify that correct input data (location and price) was used to calculate the fee.

We formally define Optimistic Payment and propose a construction based on homomorphic commitments and signature schemes that allow for efficient zero-knowledge proofs of signature possession. We prove our construction secure under standard assumptions. Finally, we present a prototype implementation on an embedded platform, and demonstrate that the cryptographic overhead of Optimistic Payment is efficient enough to be practically deployed in commercial in-car devices. Further, the fact that on-board units carry out all operations without interaction with the driver makes our system ideal in terms of usability.

The rest of the paper is organized as follows: we describe our system models and the security properties we seek in Sect. 2. Sect. 3 presents a high level description of our construction. Our prototype implementation and its evaluation are presented in Sect. 4, and we discuss some practical issues in Sect. 5. We situate our work within the landscape of proposals for privacy-friendly vehicular applications in Sect. 6, and we conclude in Sect. 7. Finally, we define the concept of Optimistic Payment in Appendix A, and describe in detail our cryptographic construction in Appendix B.

## 2 System model

PrETP employs the architecture and technologies recommended at European level [13, 20], although it could be adapted to other systems, such as [1]. The system model, illustrated in Fig. 1 (left), comprises three entities: an On-Board Unit (OBU), a Toll Service Provider (TSP), and a Toll Charger (TC). The OBU is an electronic device installed in vehicles subscribed to an ETP service, and it is in charge of collecting GPS data and calculating the fee at the end of each tax period. The TSP is the entity that offers the ETP service. It is responsible for providing vehicles with OBUs and monitor their performance and integrity. Finally, the TC is the organization (either public or private) that levies tolls for the use of roads and defines the correct use of the system. In agreement with the TC, the TSP establishes prices for driving on each of the roads. Such pricing policy can depend on the type of road (e.g., highways vs. secondary roads), its traffic density, or the time of the day (e.g., rush hours vs. the middle of the night). Additionally, prices can also depend on attributes of the vehicle or of the driver (e.g., low-pollution vehicles, or discounts for retired people). For the sake of clarity, in this work we focus on the core functionality of PrETP, and defer the discussion of practical issues to Sect. 5.

When the vehicle is driving, the OBU calculates the subfees corresponding to the trajectories according to the TSP pricing policy. At the end of each tax period, the OBU aggregates all the subfees to obtain a total fee and sends it to the TSP. This process safeguards the privacy of the driver from the TSP, the TC, or any other third party eavesdropping the communications, as no location data leaves the OBU. The privacy objectives of PrETP focus on the limitation of deliberate surveillance by any external party with limited access to the vehicle. We note that for an adversary with physical access to the vehicle it would be trivial to track it, e.g. by installing a tracking device. In order to further protect the privacy of users from adversaries that have occasional access to OBUs (e.g., mechanic, valet), all location data stored in the OBU is securely encrypted as specified in [44].

Besides preserving users' privacy, the system has to protect the interests of both TC and TSP and provide means to prevent users from committing fraud. Our threat model considers malicious drivers capable of tampering with the internal functionality of the OBU, as well as with any of its interfaces. Under these considerations, we define the security goals of our system as the detection of:

**Vehicles with inactive OBUs.** Drivers should not be able to shut down their OBUs at will to simulate they drove less.

**OBUs reporting false GPS location data.** Drivers should not be able to spoof the GPS signal and simulate a cheaper route than the actual roads on which they are driving.

**OBUs using incorrect road prices.** Drivers should not be able to assign arbitrary prices to the roads on which they are driving.

**OBUs reporting false final fees.** Drivers should not be able to report an arbitrary fee, but only the result from the correct calculations in the OBU.

Focusing on the detection of tampering rather that at its prevention allows us to consider a very simple OBU with no trusted components, reducing the production costs of the device.

In order to perform this detection, reliable information about the vehicle's whereabouts is required. We consider that the TC can perform random "spot checks" that are recorded as proof of the time and location where a vehicle has been seen. Such spot checks can be carried out by using an automatic license plate reader, a police control, or even challenging the OBUs using Dedicated Short-Range Communications (DSRC) [13]. Without loss of generality in this work we assume that the proof is gathered using an automatic license plate reader. This proof can be used to challenge the vehicle's OBU to verify its functioning. In order to be able to respond to this challenge, the OBU slices the trajectories recorded in segments, and computes the subfees corresponding to them, such that these subfees add up to the final fee transmitted to the TSP. For each segment, the TSP receives a payment tuple that consists of a commitment to location data and time, a homomorphic commitment to the subfee, and a proof that the committed subfee is computed according to the policy. These payment tuples, explained in detail in the next section, bind the reported final fee to the committed values such that the OBU cannot claim having used other locations or prices in its computations. Furthermore, they are signed by the OBU to prevent a malicious TSP from framing an honest driver.

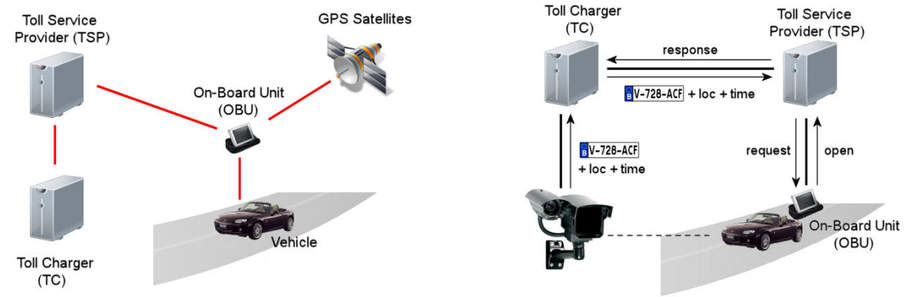



Figure 1: Entities in our Electronic Toll Pricing architecture (left.) Enforcement spot-check model (right.)

The verification process, depicted in Fig. 1 (right), is initiated when the TC gathers a proof of location of a vehicle. Then it forwards this information to the TSP, along with a request to check the correct functioning of the vehicle's OBU. To this end, the TSP challenges the OBU to open a commitment containing the location and time appearing in the proof. The TSP verifies that both challenge and response match, for instance as explained in [39], and reports to the TC whether or not the functioning of the OBU is correct. We assume that the TC (e.g., the government in the EETS architecture) is honest and does not use fake proofs to challenge OBUs.

## 3 Optimistic Payment

In this section we sketch the technical concepts necessary to understand the construction of Optimistic Payment, and we outline our efficient implementation of the protocol. For a comprehensive and more formal description of OP, we refer the reader to Appendix B.

### 3.1 Technical Preliminaries

**Signature Schemes.** A signature scheme consists of the algorithms SigKeygen, SigSign and SigVerify. SigKeygen outputs a secret key $sk$ and a public key $pk$. $\mathsf{SigSign}(sk, x)$ outputs a signature $s_x$ of message $x$. $\mathsf{SigVerify}(pk, x, s_x)$ outputs accept if $s_x$ is a valid signature of $x$ and reject otherwise. A signature scheme must be correct and unforgeable [26]. Informally speaking, correctness implies that the SigVerify algorithm always accepts an honestly generated signature. Unforgeability means that no p.p.t adversary should be able to output a message-signature pair $(x, s_x)$ unless he has previously obtained a signature on $x$.

**Commitment schemes.** A non-interactive commitment scheme consists of the algorithms ComSetup, Commit and Open. $\mathsf{ComSetup}(1^k)$ generates the parameters of the commitment scheme $params_{Com}$. $\mathsf{Commit}(params_{Com}, x)$ outputs a commitment $c_x$ to $x$ and auxiliary information $open_x$. A commitment is opened by revealing $(x, open_x)$ and checking whether $\mathsf{Open}(params_{Com}, c_x, x, open_x)$ is true. A commitment scheme has a hiding property and a binding property. Informally speaking, the hiding property ensures that a commitment $c_x$ to $x$ does not reveal any information about $x$, whereas the binding property ensures that $c_x$ cannot be opened to another value $x'$. Given two commitments $c_{x_1}$ and $c_{x_2}$ with openings $(x_1, open_{x_1})$ and $(x_2, open_{x_2})$ respectively, the additively homomorphic property ensures that, if $c = c_{x_1} \cdot c_{x_2}$, then $\mathsf{Open}(params_{Com}, c, x_1 + x_2, open_{x_1} + open_{x_2})$.

**Proofs of Knowledge.** A zero-knowledge proof of knowledge is a two-party protocol between a prover and a verifier. The prover proves to the verifier knowledge of some secret values that fulfill some statement without disclosing the secret values to the verifier. For instance, let $x$ be the secret key of a public key $y = g^x$, and let the prover know $(x, g, y)$, while the verifier only knows

$(g, y)$. By means of a proof of knowledge, the prover can convince the verifier that he knows $x$ such that $y = g^x$, without revealing any information about $x$.

### 3.2 Intuition Behind Our Construction

We consider a setting with the entities presented in Sect. 2. During each tax period $tag$, the OBU slices the trajectories of the driver in segments formed by a structure containing GPS location data and time. Additionally, this data structure can contain information about any other parameter that influences the price to be paid for driving on the segment. We represent this data structure as a tuple $(loc, time)$. The TSP establishes a function $f : (loc, time) \rightarrow \Upsilon$ that maps every possible tuple $(loc, time)$ to a price $p \in \Upsilon$. For each segment, the OBU calculates $f$ on input $(loc, time)$ to get a price $p$, and computes a payment tuple that consists of a randomized hash $h$ on the data structure $(loc, time)$, a homomorphic commitment $c_p$ to its price, and a proof $\pi$ that the committed price belongs to $\Upsilon$. The randomization of the hash is needed in order to prevent dictionary attacks to recover $(loc, time)$.

At the end of the tax period, the OBU and the TSP engage in a two-party protocol. The OBU adds the fees of all the segments to obtain a total fee $fee$. The OBU adds all the openings $open_p$ to obtain an opening $open_{fee}$. Next, the OBU composes a payment message $m$ that consists of $(tag, fee, open_{fee})$ and all the payment tuples $(h, c_p, \pi)$. The OBU signs $m$ and sends both the message $m$ and its signature $s_m$ to the TSP. The TSP verifies the signature and, for each payment tuple, verifies the proof $\pi$. Then the TSP, by using the homomorphic property of the commitment scheme, adds the commitments $c_p$ of all the payment tuples to obtain a commitment $c'_{fee}$, and checks that $(fee, open_{fee})$ is a valid opening for $c'_{fee}$.

When the TC sends the TSP a proof $\phi$ that a car was at some position at a given time, the TSP relays $\phi$ to the OBU. The OBU first verifies that the request is signed by the TC, and then it searches for a payment tuple $(h, c_p, \pi)$ for which $\mu(\phi, (loc, time))$ outputs accept. Here, $\mu : (\phi, (loc, time)) \rightarrow \{\text{accept}, \text{reject}\}$ is a function established by the TSP that outputs accept when the information in $\phi$ and in $(loc, time)$ are similar in accordance with some metric, such as the one proposed in [39]. Once the payment tuple is found, the OBU sends the number of the tuple to the TSP together with the preimage $(loc, time)$ of $h$ and the opening $(p, open_p)$ of $c_p$. The TSP checks that $(p, open_p)$ is the valid opening of $c_p$, that $(loc, time)$ is the preimage of $h$ and that $\mu(\phi, (loc, time))$ outputs accept.

Intuitively, this protocol ensures the four security properties enunciated in the previous section. Drivers cannot shut down their OBUs, nor report false GPS data as they run the risk of not having committed to a segment containing the $(loc, time)$ in the challenge $\phi$. We note that after sending $(m, s_m)$ to the TSP, OBUs cannot claim that they were at any position $(loc', time')$ different from the ones used to compute the message $m$. Similarly, OBUs cannot use incorrect road prices without being detected, as the TSP can check whether the correct price for a segment $(loc, time)$ was used once the commitments are opened. The homomorphic property ensures that the reported final fee is not arbitrary, but the sum of all the committed subfees. Moreover, by making the OBU prove that the committed prices belong to the image of $f$, we avoid that a malicious OBU could decrease the final fee by sending only one wrong commitment to a negative price in the payment message, which would give it an overwhelming probability of not being detected by the spot checks. Additionally, the fact that the OBU signs the payment message $m$ ensures that no malicious TSP can frame an OBU by modifying the received commitments, and that a malicious OBU cannot plead innocent by invoking the possibility of being framed by a malicious TSP. Similarly, the fact that the TC signs the challenge $\phi$ prevents a malicious TSP sending fake proofs to the OBU, e.g. with the aim of learning its location. Finally, the privacy of the drivers is preserved as the OBU does not need to disclose more location information than that in the payment tuple that matches the proof $\phi$ (already known to TSP).

### 3.3 Efficient Instantiation: High Level Specification

We now outline at high level our efficient instantiation of Optimistic Payment. We employ the integer commitment scheme due to Damgård and Fujisaki [15] and the CL-RSA signature scheme proposed by Camenisch and Lysyanskaya [9]. Both schemes use cryptographic keys based on special RSA modulus $n$ of length $l_n$. A commitment $c_x$ to a value $x$ is computed as $c_x = {g_0}^x {g_1}^{open_x} \pmod n$, where the opening $open_x$ is a random number of length $l_n$ and the bases $(g_0, g_1)$ correspond to the commitment public parameters. Given a public key $pk = (n, R, S, Z)$, a CL-RSA signature has the form $(A, e, v)$, with lengths $l_n$, $l_e$, and $l_v$ respectively, such that $Z \equiv A^e R^x S^v \pmod n$. To prove that a price belongs to $\Upsilon$, we use a non-interactive proof of possession of a CL-RSA signature on the price. We also employ a collision resistant hash function $H : \{0,1\}^* \rightarrow \{0,1\}^{l_c}$.

**Initialization.** The pricing policy $f : (loc, time) \rightarrow \Upsilon$, where each price $p \in \Upsilon$ has associated a valid CL-RSA signature $(A, e, v)$ generated by the TSP, the cryptographic key pair $(pk_{\text{OBU}}, sk_{\text{OBU}})$, the public key of the TSP $(n, R, S, Z)$, the public key of TC, and the public parameters $(g_0, g_1)$ of the commitment scheme are stored on the OBU. Similarly, the TSP possesses its own secret key $(sk_{\text{TSP}})$ and knows all the public keys in the system.

| | **OBU** Pay() algorithm | | **TSP** VerifyPayment() algorithm | |
|---|---|---|---|---|
| 1 | **// Main loop** | | | 1 |
| 2 | *For all* $1 \leq k \leq N$ *tuples do:* | | | 2 |
| 3 | $p_k = f(loc_k, time_k)$ | | | 3 |
| 4 | **// Hash computation** | | | 4 |
| 5 | $h_k = H((loc_k, time_k))$ | | | 5 |
| 6 | **// Commitment computation** | | | 6 |
| 7 | $open_{p_k} \leftarrow \{0,1\}^{l_n}$ | | | 7 |
| 8 | $c_{p_k} = {g_0}^{p_k} {g_1}^{open_{p_k}} \pmod n$ | | | 8 |
| 9 | **// Proof computation** | | | 9 |
| 10 | $open_w, w \leftarrow \{0,1\}^{l_n}$ | | | 10 |
| 11 | $\tilde{A} = A {g_0}^w \pmod n$ | | | 11 |
| 12 | $c_w = {g_0}^w {g_1}^{open_w} \pmod n$ | | OBUverify$(pk_{\text{OBU}}, m, s_m)$ | 12 |
| 13 | $r_\alpha \leftarrow \{0,1\}^{l_\alpha}$ | | **// Main loop** | 13 |
| 14 | $t_{c_{p_k}} = {g_0}^{r_{p_k}} {g_1}^{r_{open_{p_k}}}$ | $\xrightarrow{(m, s_m)}$ | *For all* $1 \leq k \leq N$ *tuples do:* | 14 |
| 15 | $t_Z = \tilde{A}^{r_e} R^{r_{p_k}} S^{r_v} (g_0^{-1})^{r_{w \cdot e}}$ | | $t'_{c_{p_k}} = c_{p_k}^{ch} {g_0}^{s_{p_k}} {g_1}^{s_{open_x}}$ | 15 |
| 16 | $t_{c_w} = {g_0}^{r_w} {g_1}^{r_{open_w}}$ | | $t'_Z = Z^{ch} \tilde{A}^{s_e} R^{s_{p_k}} S^{s_v} (1/g_0)^{s_{w \cdot e}}$ | 16 |
| 17 | $t = c_w^{r_e} (g_0^{-1})^{r_{w \cdot e}} (g_1^{-1})^{r_{open_{w \cdot e}}}$ | | $t'_{c_w} = c_w^{ch} {g_0}^{s_w} {g_1}^{s_{open_w}}$ | 17 |
| 18 | $ch = H(\beta \| t_{c_{p_k}} \| t_Z \| t_{c_w} \| t)$ | | $t' = C_w^{s_e} (1/g_0)^{s_{w \cdot e}} (1/g_1)^{s_{open_{w \cdot e}}}$ | 18 |
| 19 | $s_\alpha = r_\alpha - ch \cdot \alpha$ | | $ch' = H(\beta \| t'_{c_{p_k}} \| t'_Z \| t'_{c_w} \| t')? = ch$ | 19 |
| 20 | $\pi_k = (\tilde{A}, c_w, ch, s_\alpha)$ | | $s_e \in \{0,1\}^{l_e + l_c + l_z}$ | 20 |
| 21 | *End for* | | $s_{p_k} \in \{0,1\}^{l_p + l_c + l_z}$ | 21 |
| 22 | **// Fee reporting** | | *End for* | 22 |
| 23 | $fee = \sum_{k=1}^{N} p_k$ | | **// Commitment validation** | 23 |
| 24 | $open_{fee} = \sum_{k=1}^{N} open_{p_k}$ | | $c'_{fee} = \prod_{k=1}^{N} c_{p_k}$ | 24 |
| 25 | $m = [tag, fee, open_{fee}, (h_k, c_{p_k}, \pi_k)_{k=1}^{N}]$ | | $c_{fee} = {g_0}^{fee} {g_1}^{open_{fee}} \pmod n$ | 25 |
| 26 | $s_m = \text{OBUsign}(sk_{\text{OBU}}, m)$ | | $c_{fee}? = c'_{fee}$ | 26 |
| | $\alpha \in \{p_k, open_{p_k}, e, v, w, open_w, w \cdot e, open_{w \cdot e}\}$ | | | |
| | $\beta = (n \| g_0 \| g_1 \| \tilde{A} \| R \| S \| g_0^{-1} \| g_1^{-1} \| c_{p_k} \| Z \| c_w \| 1)$ | | | |

Protocol 1: Protocol between OBU and TSP during taxing phase

**Tax period.** Protocol 1 illustrates the calculations and interactions between the OBU and the TSP under normal functioning during the tax period. We denote the operations carried out by the OBU as Pay(), and the operations executed by the TSP as VerifyPayment(). While driving, the OBU collects location data and slices it in segments $(loc, time)$ according to the policy. For each of the $N$ collected segments, the OBU generates a payment tuple $(h_k, c_{p_k}, \pi_k)$. This iterative step is broken down in lines 1 to 21 in Protocol 1. The most resource consuming operation is the computation of $\pi_k$, which proves the possession of a valid CL-RSA signature on the price $p_k$ (lines 9 to 20). The length of the random values used in this step is specified in Appendix B.2. At the end of the tax period the OBU generates and signs the payment message $m$ including the tag $tag$, the total fee, the opening $open_{fee}$, and all the payment tuples $(h_k, c_{p_k}, \pi_k)$, lines 22 to 26. Finally it sends $(m, s_m)$ to the TSP.

Upon reception of a payment message, the TSP executes the VerifyPayment() algorithm. First the TSP verifies the signature $s_m$ using the OBU's public key $pk_{\text{OBU}}$. Next, it proceeds to the verification of the proof $\pi_k$ included in each of the $N$ payment tuples contained in $m$, lines 13 to 22. In each iteration it performs a series of modular exponentiations, and uses the intermediate results to compute the hash $ch'$. Then, it checks whether $ch'$ is the same as the value $ch$ contained in $\pi_k$. If this verification, together with the two range proofs in lines 20 and 21, is successful, the TSP is convinced that all the prices $p_k$ used by the OBU are indeed a valid image of $f$. Finally, the TSP validates the commitments $c_{p_k}$ to ensure that the aggregation of all subfees add up to the final fee (lines 24 to 26). For this, it calculates $c'_{fee}$ as the product of all commitments $c_{p_k}$, and computes the com-

mitment $c_{fee}$ using the values $fee$ and $open_{fee}$ provided by the OBU. If both values are the same, the TSP is convinced that the final fee reported by the OBU adds up to the sum of all subfees reported in the payment tuples.

**Proof Challenge.** We denote as OBUopen() and Check() the algorithms carried out by the OBU and the TSP, respectively, when the former is challenged with $\phi$. When running the OBUopen() algorithm, the OBU searches for the pre-image $(loc_k, time_k)$ of a hash $h_k$ containing the location and time satisfying $\phi$, and sends this information to the service provider along with the price $p_k$ and the opening $open_{p_k}$.

Upon reception of this message, the TSP executes the Check() algorithm. First, it verifies whether the segment $(loc_k, time_k)$ actually contains the location in $\phi$. Then, it computes the value $h'_k = H(loc_k, time_k)$ and checks whether the OBU had committed to this value in one of the payment tuples reported during the tax period. Lastly, the TSP uses $open_{p_k}$ to open the commitment $c_{p_k}$ and verifies whether $p'_k = f(loc_k, time_k)$ equals the price $p_k$ reported by the OBU during the OBUopen() algorithm. If all verifications succeed, the TSP is convinced that the location data used by the OBU in the fee calculation and the price assigned by the OBU to the segment $(loc_k, time_k)$ are correct.

# 4 PrETP Evaluation

In this section we evaluate the performance of PrETP. We start by describing the test scenario and both our OBU and TSP prototypes. Next, we analyze the performance of the prototypes for different configuration parameters. Finally, we study the communication overhead in PrETP, and compare it to existing ETP systems.

## 4.1 Test Scenario

**Policy model.** The first step in the implementation of PrETP consists in specifying a policy model in the form of the mapping function $f : (loc, time) \rightarrow \Upsilon$. We decide to follow the same criteria as currently existing ETP schemes [36], i.e., road prices are determined by two parameters: type of road and time of the day. More specifically, we define three categories of roads ('highway', 'primary', and 'others') and three time slots during the day. For each of the possible nine combinations we assign a price per kilometer $p$ and we create a valid signature $(A, e, v)$ using the TSP's secret key. We note that the choice of this policy is arbitrary and that PrETP, as well as OP, can accommodate other price strategies.

**Location data.** We provide the OBU with a set of location data describing a real trajectory of a vehicle . These data are obtained by driving with our prototype for one hour in an urban area, covering a total distance of 24 kilometers. We note that such dataset is sufficient to validate the performance of PrETP, since results for different driving scenarios (e.g., faster or slower) can easily be extrapolated from the results presented in this section.

**Parameters of the instantiation.** The performance of OP depends directly on the length of the protocol instantiation parameters, and in particular, on the size of the cryptographic keys of the entities ($l_n$). In our experiments we consider three case studies: medium security ($l_n$ = 1024 bits), high security ($l_n$ = 1536 bits), and very high security ($l_n$ = 2048 bits). The value $l_p$ is determined by the length of the prices $p$, which in turn determines the value of $l_e$. Therefore, both lengths are constant for all security cases. The value of $l_v$ varies depending on the value of $l_n$. Finally, the rest of parameters ($l_h$, $l_r$, $l_z$, and $l_c$) are set as the output length of the chosen hash function primitive (see Sect. 4.2). These lengths determine the size of the random numbers generated in line 13 in Protocol 1 (see Appendix B for a detailed explanation). Table 1 summarizes the parameter lengths considered for each security level.

Table 1: Length of the parameters (in bits)

| Parameter | $l_n$ | $l_e$ | $l_v$ | $l_p$ | $l_r$,$l_h$,$l_z$,$l_c$ |
|---|---|---|---|---|---|
| **Normal Sec.** | 1 024 | 128 | 1 216 | 32 | 160 |
| **High Sec.** | 1 536 | 128 | 1 728 | 32 | 160 |
| **Very high Sec.** | 2 048 | 128 | 2 240 | 32 | 160 |

**OBU Platform.** In order to make our prototype as realistic as possible, we implement PrETP using as starting point the embedded design described in [4], which performs the conversion of raw GPS data into a final fee internally. We extend and adapt this prototype with the functionalities of OP to make it compatible with PrETP.

At high-level, the elements of our OBU prototype [4] are: a processing unit, a GPS receiver, a GSM modem, and an external memory module. We use as benchmark the Keil MCB2388 evaluation board [30], which contains an NXP LPC2388 [34] 32-bit ARM7TDMI [2] microcontroller. This microcontroller implements a RISC architecture, it runs at 72 MHz, and it offers 512 Kbytes of on-chip program memory and 98 Kbytes of internal SRAM. As external memory, we use an off-the-shelf 1 GByte SD Card connected to the microcontroller. Finally, we use the Telit GM862-GPS [43] as both GPS receiver and GSM modem.

As our platform does not contain any cryptographic coprocessors, we implement all functionalities exclusively in software. Note that although we could easily add a hardware coprocessor (e.g., [35]) to the prototype in order to carry out the most expensive cryptographic computations, we choose the option that minimizes the production costs of the OBU. Besides, this approach allows us to identify the bottlenecks in the protocol implementation, leaving the door open to hardware-based improvements if needed.

We have constructed a cryptographic library with the primitives required by our instantiation of the OP protocol, namely: i) a modular exponentiation technique, ii) a one-way hash function, and iii) a random number generator. For the first primitive we use the ACL [5] library, a collection of arithmetic and modular routines specially designed for ARM microcontrollers. As hash function we choose RIPEMD-160 [22], with an output length $l_h$ of 160 bits. As our platform does not provide any physical random number generator, we use the Salsa20 [6] stream cipher in keystream mode as third primitive. We note that a commercial OBU should include a source of true randomness.

In order to keep the OBU flexible and easily scalable, we arrange data in different memory areas depending on their lifespan. Long-term parameters ($pk_{\text{OBU}}$, $sk_{\text{OBU}}$, $pk_{\text{TSP}}$, commitment parameters) are directly embedded into the microcontroller's program memory, while short-term parameters (payment tuples, $(loc, time)$ segments) and updatable parameters (digital road map, policy $f$) are stored separately on the SD Card. We note that our library provides a byte-oriented interface with the SD Card, resulting in a considerable overhead when reading/writing values.

**TSP Platform.** We implement our TSP prototype on a commodity computer equipped with an Intel Core2 Duo E8400 processor at 3 GHz, and 4 Gbyte of RAM. We use C as programming language, and the GMP [25] library for large-integer cryptographic operations.

## 4.2 Performance Evaluation

**OBU performance.** The most time-consuming operations carried out by the OBU during the taxing phase are the Mapping() algorithm and the Pay() algorithm. The Mapping() algorithm is executed every time a new GPS string is available in the microcontroller. Its function is to search in the digital road map the type of road given the GPS coordinates. When the vehicle drives for a kilometer, the OBU maps the segment to the adequate price $p_k$ as specified in the policy. At this point, the Pay() algorithm is executed in order to create the payment tuple. For each segment, the OBU generates: i) a hash value $h_k$ of the location data, ii) a commitment $c_{p_k}$ to the price $p_k$, and iii) a proof $\pi_k$ proving that the price $p_k$ is genuinely signed by the TSP (and thus belongs to the image of $f$). To protect users' privacy we also require that no sensitive data is stored in the SD Card in plaintext form. For this purpose we use the AES [33] block cipher in CCM mode [23] with a key length of 128 bits. We denote this operation as $E_k$. At the end of the taxing phase, the OBU adds all the prices $p_k$ mapped to each segment to obtain the fee, and all the openings $open_k$ to obtain $open_{fee}$. Finally, the OBU constructs and signs the payment message $m$ and sends it to the TSP.

As it does not involve the key, the computing time of the Mapping() algorithm is independent of the security scenario. Further, this time only depends on the duration of the trip and is independent of the speed of the vehicle: the Mapping() algorithm is always executed 3 600 times per hour, taking a total of 839.11 seconds in our prototype. However, for each of the segments this time can vary depending on the number of points that have to be processed, i.e., depending on the speed of the vehicle. In our experiments it requires 76.10 seconds for the longest segment, i.e., the one where the vehicle spent more time to drive one kilometer and thus $(loc_k, time_k)$ contains the larger number of points.

Similarly, the execution time for $h_k$ and $E_k$ depends exclusively on the length of the segments $(loc_k, time_k)$, as it is proportional to the number of GPS points in the segments. The amount of points per segment varies not only with the average speed of the car but also depending on the length of the segments defined in the pricing policy. In our experiments, computing $h_k$ and $E_k$ take 0.08 seconds and 0.43 seconds, respectively, for the shortest and the longest segments. For the Mapping() algorithm and both $h_k$ and $E_k$ operations, more than 90% of the time is spent in the communication with the SD card.

On the other hand, the execution time for $c_{p_k}$ and $\pi_k$ is constant for all segments, as it does not depend on the length of a particular slice (see lines 6 to 20 in Protocol 1). In order to calculate $c_{p_k}$, the OBU needs to generate a random opening $open_{p_k}$ and perform two modular exponentiations and a modular multiplication. The computation of $\pi_k$ involves the generation of ten random numbers and a hash value, and the execution of fourteen modular exponentiations, nine modular multiplications, eight additions, and eight multiplications. The bottleneck of both operations is determined by the modular operations. Although we could take advantage of fixed-base modular exponentiation techniques, we choose to use multi-exponentiations algorithms [18], which have less storage requirements. Multi-exponentiation based algorithms, which compute values of the form $a^b c^d \pmod n$ in one step, allow us to considerably speed up the process. The average execution times for computing $c_{p_k}$ are 0.76 seconds, 2.25 seconds, and 5.69 seconds for medium, high, and very high security respectively. For $\pi_k$, these times are 6.20 seconds, 19.45 seconds, and 41.64 seconds, respectively.

Table 2 summarizes the timings for all OBU operations and routines for a journey of one hour. We note

Table 2: Execution times (in seconds) for an hour journey of 24 km, for all possible security scenarios.

| Algorithm | | Medium Security | | High Security | | Very high Security | |
|---|---|---|---|---|---|---|---|
| | | **Segment** | **Full trip** | **Segment** | **Full trip** | **Segment** | **Full trip** |
| Mapping() | | 76.10 s | 839.11 s | 76.10 s | 839.11 s | 76.10 s | 839.11 s |
| Pay() | | 7.88 s | 183.91 s | 22.13 s | 528.47 s | 47.79 s | 1 143.30 s |
| | $h_k$ | 0.08 s | 1.08 s | 0.08 s | 1.08 s | 0.08 s | 1.08 s |
| | $E_k$ | 0.43 s | 6.35 s | 0.43 s | 6.35 s | 0.43 s | 6.35 s |
| | $c_{p_k}$ | 0.76 s | 18.19 s | 2.25 s | 54.08 s | 5.69 s | 136.82 s |
| | $\pi_k$ | 6.20 s | 158.09 s | 19.45 s | 466.96 s | 41.64 s | 999.05 s |

that, even when 2048-bit RSA keys are used, the OBU can perform all operations needed to create the payment tuples in real time. While the trip lasted one hour, the Mapping() and Pay() algorithms only required 1 982.41 seconds. The computation time is dominated by the Pay() algorithm, which depends on the number of GPS strings in each segment $(loc, time)$. This number varies with the speed of the vehicle and the pricing policy. If a vehicle is driving at a constant speed, policies that establish prices for small distances result in segments containing less GPS points than policies that consider long distances. Similarly, given a policy fixing the size of the segments, driving faster produces segments with less points than driving slower. In both cases, $\pi_k$ has to be computed fewer times and the Pay() algorithm runs faster. Thus, the policy can be used as tuning parameter to guarantee the real-time operation of the OBU.

Using the values in Table 2, for each of the levels of security we can calculate the time our OBU is idle – in our case $(3\,600 - 839.11)$ seconds, with 839.11 seconds being the time required by the Mapping() algorithm. Then, considering our current policy, we can estimate the number of times the Pay() algorithm could be executed, which in turn represents the number of kilometers that could have been driven by a car in one hour, i.e., the average speed of the car. For normal security, our OBU could operate in real time even if a vehicle was driving at 350 km/h. This speed decreases to 124 km/h when 1536-bit keys are used, and to 57 km/h if the keys have length 2048 bits. Only when using high security parameters our OBU would have problems to operate in the field. However, as mentioned before, including a cryptographic coprocessor in the platform would suffice to solve this problem whenever high security is required. Moreover, in our tests we consider a worst-case scenario in which all GPS strings are processed upon reception. In fact, processing fewer strings would suffice to determine the location of the vehicle. As the execution time required by the Mapping() algorithm would decrease linearly, OBUs would be able to support higher vehicle speeds.

In the OBUopen() algorithm, only executed upon request from TC, the OBU searches its memory for a segment $(loc, time)$ in accordance to the proof sent by the TSP. Here, the time accuracy provided by the GPS system is used to ensure synchronization between the data in $\phi$ and the segment $(loc, time)$. The main bottleneck of this operation is the decryption of the location data corresponding to the correct segment. On average, our prototype can decrypt such a segment in 0.27 seconds.

**TSP performance.** The most consuming task the TSP must perform corresponds to the VerifyPayment() algorithm, which has to be executed each time the TSP receives a payment message. This algorithm involves three operations: the verification of the proof $\pi_k$ for each segment, the multiplication of all commitments $c_{p_k}$ to obtain $c_{fee}$, and the opening of $c_{fee}$ in order to check whether it corresponds to the reported final fee. The most costly operation is the verification of $\pi_k$, in particular the calculation of the parameters $(t'_{c_m}, t'_Z, t'_{c_w}, t')$ which requires a total of eleven modular exponentiations (lines 14 to 22 in Protocol 1).

Table 4.2 (left) shows the performance of the VerifyPayment() algorithm for each of the considered security levels when segments have length one kilometer. We also provide an estimation of the time required to process all the proofs sent by OBU during a month, assuming that a vehicle drives an average of 18 000 km per year (1 500 km per month).

These results allow us to extrapolate the number of OBUs that can be supported by a single TSP in each security scenario for different segment lengths. Intuitively, the capacity of TSP increases when segments are larger, as the payment messages contain fewer proofs $\pi_k$. The number of OBUs supported by a single TSP is presented in Table 4.2 (right). For a segment length of 1 km, the TSP is able to support 164 000, 58 000, and 29 000 vehicles depending on the chosen security level. Even when $l_n$ is 2048 bits, only 36 servers are needed to accommodate one million OBUs. This number can be reduced by parallelizing tasks at the server side, or by using fast cryptographic hardware for the modular exponentiations.



Table 3: Timings (in seconds) for the execution of VerifyPayment() in TSP (left). Number of OBUs supported by a single TSP (right).

| VerifyPayment() | **Segment** | **One Month** |
|---|---|---|
| **Medium Sec.** | 0.0105 s | 15.750 s |
| **High Sec.** | 0.0295 s | 44.250 s |
| **Very high Sec.** | 0.0587 s | 88.050 s |

| **Segment size** | **Medium Sec.** | **High Sec.** | **Very high Sec.** |
|---|---|---|---|
| **0.5 km** | 82 000 | 29 000 | 14 000 |
| **0.75 km** | 123 000 | 43 000 | 22 000 |
| **1 km** | 164 000 | 58 000 | 29 000 |
| **2 km** | 329 000 | 117 000 | 58 000 |
| **3 km** | 493 000 | 175 000 | 88 000 |

## 4.3 Communication overhead

We now compare the communication overhead of PrETP with respect to straightforward ETP implementations and VPriv [39]. Both in straightforward ETP implementations and in VPriv the OBU sends all GPS strings to the TSP. Let us consider that vehicles drive 1 500 km per month at an average speed of 80 km/h. Then, transmitting the full GPS information to the the TSP requires 2.05 Mbyte (considering a shortened GPS string of 32 bytes containing only latitude, longitude, date and time). VPriv requires more bandwidth than straightforward ETP systems, as extra communications are necessary to carry out the interactive verification protocol (see Sect. 6). Using PrETP, the communication overhead comes from the payment tuples that must be sent along with the fee. For each segment, the OBU sends the payment tuple $(h, c_p, \pi)$ to the TSP. When sent uncompressed, this implies an overhead of approximately 1.5 Kbyte per segment, i.e., less than 2 Mbyte per month, for medium security ($l_n$=1024 bits). Additionally, less than 50 Kbyte have to be sent occasionally to respond a verification challenge after a vehicle has been seen at a spot check. We believe this overhead is not excessive for the additional security and privacy properties offered by PrETP.

The communication overhead in PrETP is dominated by the payment message $m$ sent by the OBU to the TSP, the length of which depends on the number of segments covered by the driver. Therefore, the segment length can be seen as a parameter of the system that tunes the tradeoff between privacy and communication overhead. The smaller the segments, the larger the communication overhead, because more tuples ($h_k$, $c_{p_k}$, $\pi_k$) need to be sent. Allowing larger segments reduces the communication cost but also reduces privacy because the OBU must disclose a bigger segment when responding a verification challenge.

Further, the communication overhead can be almost eliminated by having the OBU sending only the hash of the payment message at the end of each tax period and leave the correct operation verification subject to random checks. Following the spirit of the random "spot checks" used for checking the input and prices, the OBUs could occasionally be challenged to prove its correct functioning by sending the payment message corresponding to the preimage of the hash sent at the end of a random tax period.

# 5 Discussion

**Practical issues.** Our OP scheme allows the OBU to prove its correct operation to the TSP while revealing a minimum amount of information. Nevertheless, we note that fee calculation is not flexible. The reason is that the OBU should store signatures created by the TSP on all the prices that belong to $\mathrm{Im}(f)$, and thus, for the sake of efficiency, we need to keep $\mathrm{Im}(f)$ small. For this purpose, in our evaluation $f$ is only defined for trajectory segments of a fixed length (one kilometer) and of a fixed road type. There are two obvious cases in which this feature is problematic: when a vehicle has driven a non-integer amount of kilometers, and when one of the segments contains pieces of roads with different cost (e.g., when a driver leaves the highway entering a secondary road). In both cases the OBU cannot produce a payment tuple because it does not have the signature by the TSP on the price of the segment.

There are two possible solutions to these issues. A first option would be to solve them at contractual level. The policy designed by the TSP could include clauses that indicate how to proceed when these conflicts arise. For instance, in the first case the TSP could dictate that the driver must pay for the whole kilometer, and in the second case the policy could be that the price corresponds to the cheapest of the roads, or to the most expensive. We note that these decisions do not conflict with the general purpose of the system: congestion control, as in all cases, on average, drivers will pay proportionally to their use of the roads. The second option would be to change the way the OBU proves that the committed prices belong to $\mathrm{Im}(f)$. In the construction proposed in Sect. 3, the OBU employs a set membership proof, based on proving signature possession, to prove that the committed prices belong to the finite set $\mathrm{Im}(f)$. Alternatively, we can define $\mathrm{Im}(f)$ as a range of (positive) prices, and let the OBU use a range proof to prove that the committed

prices belong to $\mathrm{Im}(f)$. Since now $\mathrm{Im}(f)$ is much bigger, $f$ can be defined for segments of arbitrary length that include several types of road. We outline a construction that employs range proofs in the extended version of this work [3].

Another issue is that our OP scheme does not offer protection against OBUs that do not reply upon receiving a verification challenge. In this case, the TSP should be able to demonstrate to the TC that the OBU is misbehaving. To permit this, the TSP can delegate to the TC the verification of the "spot-check", i.e, the TSP sends the payment message $m$ and the signature $s_m$ to the TC, and the TC interacts with the OBU (electronically, or by contacting the driver through some other means) to verify that $m$ is valid.

Although in Sect. 2 we mentioned that the cost associated with roads could depend on attributes of the driver (e.g., retired users may get discounts) or on attributes of the car (e.g., ecological cars may have reduced fees), the pricing policy used by our prototype is rather inflexible. We note that this is a limitation of our prototype and that PrETP can support more flexible policies. For instance, the TSP can apply discounts to the total fee reported by the OBU, without the knowledge of fine grained location data. Further, the system model in this work considers only one service provider. However, the European legislation [13, 20] points out that several TSPs may provide services in a given Toll Charger domain. PrETP can be trivially extended to this setting.

**Production cost.** Our OBU prototype, constructed with off-the-shelf components, demonstrates that a system like PrETP can be built at a reasonable cost [1]. Although the security of our Optimistic Payment scheme does not rely on any countermeasure against physical attacks by drivers, for liability reasons it is desirable to use OBUs with a certain level of tamper resistance. Nevertheless, we note that on-board units in the market [36, 42] already rely on tamper resistance. Further, secure remote firmware updates are also required in privacy invasive designs, and additional updates in PrETP containing new maps and policies can be considered occasional.

**Privacy.** Although we protect the privacy of the users by keeping the location data in the client domain and exploiting the hiding property of cryptographic commitments, there exist a few sources of information available to the TSP. First, as in many other services, users in PrETP must subscribe to the service by revealing their identity, and most likely their home address, to the TSP. Second, the final fee and all the commitments (which indicate the number of kilometers driven), must be sent to the TSP at the end of each tax period. Decoding techniques (e.g., [16]) using these data could be employed by the TSP to infer the trajectories followed by a vehicle by inspecting the possible combination of prices per kilometers that could have generated the total fee. A possible solution to this problem consists in giving users the possibility to send data associated to dummy segments. For this, a price $p$ zero should be included in the pricing policy so that it does not imply any cost for the drivers when aggregating the homomorphic commitments, and that the proofs $\pi_k$ are still accepted by the TSP. The downside of this approach is that it introduces an overhead in both the processing of the OBU and the communication link with the TSP. Apart from this, subliminal channels in the communication or the encryption schemes must be avoided, e.g., by proving a true physical randomness source in the OBU (see [44] for further discussion on the topic).

**Legal Compliance.** We build on the analysis presented in [44] and discuss the compliance of PrETP with European Legislation. With regard to data processing, the data controller (Art.6.2. in [13]) has to abide by principles found in the Data Protection Directive 95/46/EC [21] (DPD) in Art. 6.1, 16 and 17. We use these principles to assess compliance of the proposed architecture since these principles have been further specified in the other provisions of the DPD. We only look at the principles of direct interest for this paper which are that i) the data must be adequate, relevant and not excessive, ii) kept accurate and up to date, iii) the data should be processed in a secure and confidential manner and iv) data should not be kept longer than necessary. Firstly, data must be kept accurate and up-to-date (Art. 6.1(d) in [21]). In PrETP the OBU commits to location data and to its price when reporting the final fee. These commitments do not reveal any details on the location or the price calculation. Given that the controller is only allowed to process the data adequate, relevant and not excessive for the provision of the service (Art. 6.1(c) in [21]), this seems a good solution to the problem. The TC and the TSP should know that the information given by the user is correct but the information that the commitment covers is not needed for PrETP [28, 38]. The commitments implemented in PrETP are designed to guarantee that the OBU sends out the correct data without putting all the user's data in the hands of the TSP or the TC. The TC might want to execute checks at certain points in time to verify the veracity of these commitments and sends "spot-checks" to the TSP, which interacts with the OBU for the sake of verification. Only at those times will more data be disclosed because then it is required to know the information the commitment is based on to know whether the commitment is reliable. Data used for verification will however only be kept when an infringement is found. If there is no infringement, the data will not be kept in accordance with data protection principles (Art. 6.1(E) in [28, 38]). Secondly, the processing must be secure and confidential as stated by Art. 16-17 in [21]. A positive step of PrETP in this regard is keeping all the data inside the OBU and the applied algorithms to protect these data [28, 38]. The algorithms presented in this work are designed to reconcile the conflicting interest of the users and the TSP, while protecting the user from excessive data processing (note that the data set in road tolling could be potentially quite comprehensive – Art. 7, Annex VI in [13]) ). This criterion may be the most important in a road tolling setting.

[1] The cost of our prototype amounts to $500; such a number would be drastically reduced in a mass-production scenario.

## 6 Related work

A privacy-friendly architecture for ETP in which location data is not revealed to the service provider was presented in [44], and its viability was shown in [4]. However, the design by [44] does not take into account that the TSP and the TC need to check the correctness of the operations carried out in the on-board unit jeopardizing its applicability to real world scenarios.

Another line of research has focused on the design of secure multi-party protocols between the TSP and the OBUs that allow TSPs to compute the total fee and detect malicious OBUs while protecting location privacy. Solutions proposed in [8, 7, 40] resort to general reductions for secure multi-party computation and are very inefficient. A more efficient protocol, VPriv, was proposed in [39]. The basic idea consists in sending the location data generated by a driver sliced into segments to the TSP, in such a way that it remains hidden among segments from multiple drivers. Then the TSP calculates the subfees (fees of small time periods that add to the final fee) of all segments and returns them to all OBUs. Each OBU uses this information to compute its total fee and, without disclosing any location data, proves to the TSP that the total fee is computed correctly, i.e., by only using the subfees that correspond to the location data input by this particular OBU. Moreover, in order to prevent malicious users from spoofing the GPS signal to simulate cheaper trips, VPriv has an out-of-band enforcement mechanism. This mechanism is based on the use of random spot checks that demonstrate that a vehicle has been at a location at a time (e.g., a photograph taken by a road-side radar). Given this proof, the TSP challenges the OBU to prove that its fee calculation includes the location where the vehicle was spotted.

The protocol proposed in [39] has several practical drawbacks. First, it requires vehicles to send anonymous messages to the server (e.g., by using Tor [19]) imposing high additional costs to the system. Second, their protocol only avoids leaking any additional information beyond what can be deduced from the anonymized database. As the database contains path segments, the TSP could use tracking algorithms to recover paths followed by the drivers [29, 27, 32] and infer further information about them. Third, the scalability of the system is limited by the complexity of the protocol on the client side, as it depends on the number of drivers in the system. Practical implementations require simplifications such as partitioning the set of vehicles into smaller groups, thus reducing the anonymity set of the drivers. Fourth, VPriv only uses spot checks to verify correctness of the location, and thus needs an extra protocol to verify the correct pricing of segments. This extra protocol produces an overhead both in terms of computation and communication complexity.

Our solution, similar to PriPAYD [44], does not require messages between the OBU and the TSP to be anonymous as the computation of the fee is made locally and no personal data is sent to the provider. Thus, no database of personal data is created and we do not need to rely on database anonymization techniques to ensure users' privacy. Further, the OBU's operations depend only on the data it collects, independently of the number of vehicles in the system. Finally, our protocol can be integrated into a stand-alone OBU without the need of external devices to carry out the cryptographic protocols.

To the best of our knowledge, the only protocol that so far employs spot checks to verify both correctness of the location and of the fee calculation is due to Jonge and Jacobs [17]. In this solution, OBUs commit to segments of location data and its corresponding subfees when reporting the total fee to the TSP. They employ hash functions as commitments. Upon being challenged to ratify the information in the spot check, OBUs must provide the hash pre-image of the corresponding segment, and demonstrate that indeed the location was used to compute the final fee.

Jonge and Jacobs' protocol is limited by the fact that using hash-based commitments one cannot prove that the commitments to the subfees add to the total fee. As solution, they propose that the OBU also commits to the subfees corresponding to bigger time intervals following a tree structure. Each tax period is divided into months, each month is divided into weeks, and so forth, and subfees for each month, week, day,... are calculated and committed. Then, instead of asking the OBU to open only one commitment containing the instant specified in TC's proof, the TSP asks the OBU to open all the commitments in the tree that include that instant. This indeed proves that the sum is correct at the cost of revealing much more information to the TSP.

PrETP avoids this information leakage. The reason is that, in our OP scheme, commitments are homomorphic and thus allow TSP to check that the commitments to the subfees add to the total fee without additional data. The

use of homomorphic commitments was also proposed and briefly sketched in [17]. However, their scheme does not prevent the OBU from committing to a "negative" price, which would give a malicious OBU the possibility of reducing the final fee by sending only one wrong commitment, thus with an overwhelming probability of not being detected by the spot checks.

## 7 Conclusion

The revelation of location data in Electronic Toll Pricing (ETP) systems, besides conflicting with the users' right to privacy, can also pose inconveniences and extra investments to service providers as the law demands that personal data is stored and processed under strong security guarantees [21]. Furthermore, it has been shown [31] that security and privacy concerns are among the main reasons that discourage the use of electronic communication services. Recent research [45] demonstrates that users confronted to a prominent display of private information not only prefer service providers that offer better privacy guarantees but also are willing to pay higher prices to utilize more privacy protective systems. Consequently, it is of interest for service providers to deploy systems where the amount of location information that users need to disclose is minimized.

As ETP systems are becoming increasingly important [13, 1], it is a challenge to implement them respecting both the users' privacy and the interest of the service provider. Previous work relied on too expensive solutions, or on unrealistic requirements, to fulfill both properties. In this work we have presented PrETP, an ETP system that allows on-board units to prove that they operate correctly leaking the minimum amount of information. Namely, upon request of the service provider, on-board units can attest that the input location data for the calculation of the fee is authentic and has not been tampered with. For this purpose we proposed a new cryptographic protocol, Optimistic Payment, that we define, construct and prove secure under standard assumptions. For this protocol, we also provide an efficient instantiation based on known secure cryptographic primitives.

We have performed a holistic analysis of PrETP. Besides the security analysis, we have built an on-board unit prototype on an embedded platform, as well as a service provider prototype on a commodity computer, and we have thoroughly tested the performance of both using real world collected data. The result of our experiments confirms that our protocol can be executed in real time in an on-board unit constructed with off-the-shelf components. Finally, we have analyzed the legal compliance of PrETP under the European Law framework and conclude that it fully supports the Data Protection Directive principles.

**Acknowledgements.** The authors want to thank M. Peeters and S. Motte for early valuable discussions, and G. Danezis and C. Diaz for their editorial suggestions that greatly improved the readability of the paper. We thank B. Gierlichs for driving us around to collect the data used in our experiments. C. Troncoso and A. Rial are research assistants of the Fund for Scientific Research in Flanders (FWO). This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State, by the Flemish IBBT NextGenITS project, by the European Commission under grant agreement ICT-2007-216676 ECRYPT NoE phase II, and by K.U. Leuven-BOF (OT/06/40). The information in this document reflects only the author's views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

## References

[1] AB 744 (Torrico) Authorize a BayArea Express Lane Network to Deliver Congestion Relief and PublicTransit Funding with No NewTaxes, August 2009.

[2] ARM. ARM7TDMI technical reference manual, revision: r4p3. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0234b/DDI0234.pdf`, 2004.

[3] J. Balasch, A. Rial, C. Troncoso, B. Preneel, I. Verbauwhede, and C. Geuens. Privacy-preserving electronic traffic pricing using optimistic payments. COSIC internal report, K.U. Leuven, 2010.

[4] J. Balasch, I. Verbauwhede, and B. Preneel. An embedded platform for privacy-friendly road charging applications. In *Design, Automation and Test in Europe (DATE 2010)*, pages 867–872. IEEE, 2010.

[5] J. Ban. Cryptographic library for ARM7TDMI processors. Master's thesis, T.U. Kosice, 2007.

[6] D. Bernstein. Salsa20. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/025, 2005.

[7] A. Blumberg and R. Chase. Congestion privacy that respects "driver privacy". In *ITSC*, 2005.

[8] A. Blumberg, L. Keeler, and A. Shelat. Automated traffic enforcement which respects driver privacy. In *ITSC*, 2004.

[9] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In *In SCN 2002, volume 2576 of LNCS*, pages 268–289. Springer, 2002.

[10] J. Camenisch and M. Stadler. Proof systems for general statements about discrete logarithms. Technical Report TR 260, Institute for Theoretical Computer Science, ETH Zürich, March 1997.

[11] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[12] D. Chaum and T. Pedersen. Wallet databases with observers. In *CRYPTO '92*, volume 740 of *LNCS*, pages 89–105, 1993.

[13] Commission Decission of 6 October 2009 on the definition of the European Electronic Toll Service and its technical elements, 2009.

[14] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Y. Desmedt, editor, *CRYPTO*, volume 839 of *LNCS*, pages 174–187. Springer, 1994.

[15] I. Damgård and E. Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In Y. Zheng, editor, *ASIACRYPT*, volume 2501 of *LNCS*, pages 125–142. Springer, 2002.

[16] G. Danezis and C. Diaz. Space-efficient private search with applications to rateless codes. In Sven Dietrich and Rachna Dhamija, editors, *Financial Cryptography*, volume 4886 of *LNCS*, pages 148–162. Springer, 2007.

[17] W. de Jonge and B. Jacobs. Privacy-friendly electronic traffic pricing via commits. In P. Degano, J. Guttman, and F. Martinelli, editors, *Formal Aspects in Security and Trust*, volume 5491 of *LNCS*, pages 143–161. Springer, 2008.

[18] V. S. Dimitrov, G. A. Jullien, and W. C. Miller. Complexity and fast algorithms for multiexponentiations. *IEEE Transactions on Computers*, 49(2), 2000.

[19] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, pages 303–320. USENIX, 2004.

[20] Directive 2004/52/EC of the European Parliament and of the Council of 29 April 2004 on the interoperability of electronic road toll systems in the Community, 2004.

[21] Directive 95/46/EC of the European parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data, 1995.

[22] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strengthened version of RIPEMD. In Dieter Gollmann, editor, *FSE*, volume 1039 of *LNCS*, pages 71–82. Springer, 1996.

[23] Morris Dworkin. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality. NIST special publication 800-38c, National Institute for Standards and Technology, 2004.

[24] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. Odlyzko, editor, *CRYPTO*, volume 263 of *LNCS*, pages 186–194. Springer, 1986.

[25] GMP. The GNU Multi-precision Library. `http://gmplib.org/`.

[26] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[27] M. Gruteser and B. Hoh. On the anonymity of periodic location samples. In D. Hutter and M. Ullmann, editors, *SPC*, volume 3450 of *LNCS*, pages 179–192. Springer, 2005.

[28] J. H. Hoepman. Follow that car! over de mogelijke privacy gevolgen van rekeningrijden, en hoe die te vermijden. *Privacy & Informatie*, 5(11):225–230, 2008.

[29] B. Hoh, M. Gruteser, H. Xiong, and A. Alrabady. Enhancing security and privacy in traffic-monitoring systems. *IEEE Pervasive Computing*, 5(4):38–46, 2006.

[30] Keil. MCB2300 Evaluation Board Family.

[31] P. Koargonkar and L. Wolin. A multivariate analysis of web usage. *Journal of Advertising Research*, pages 53–68, March/April 1999.

[32] J. Krumm. Inference attacks on location tracks. In A. LaMarca, M. Langheinrich, and K. Truong, editors, *Pervasive*, volume 4480 of *LNCS*, pages 127–143. Springer, 2007.

[33] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197)*. National Institute of Standards and Technology, November 2001.

[34] NXP Semiconductors. LPC23xx User Manual.

[35] NXP Semiconductors. SmartMX P5xC012/020/024/037/052 family. Secure contact PKI smart card controller.

[36] Octo Telematics S.p.A. `http://www.octotelematics.com/`.

[37] T. Okamoto. An efficient divisible electronic cash scheme. In D. Coppersmith, editor, *CRYPTO*, volume 963 of *LNCS*, pages 438–451. Springer, 1995.

[38] International Working Group on Data Protection in Telecommunications. Report and Guidance on Road Pricing, "Sofia Memorandum".

[39] R. Popa, H. Balakrishnan, and A. Blumberg. VPriv: Protecting privacy in location-based vehicular services. In *Proceedings of the 18th Usenix Security Symposium*, August 2009.

[40] S. Rass, S. Fuchs, M. Schaffer, and K. Kyamakya. How to protect privacy in floating car data systems. In V. Sadekar, P. Santi, Y. Hu, and M. Mauve, editors, *Vehicular Ad Hoc Networks*, pages 17–22. ACM, 2008.

[41] C. Schnorr. Efficient signature generation for smart cards. *Journal of Cryptology*, 4(3):239–252, 1991.

[42] STOK Nederland BV. `http://www.stok-nederland.nl/`.

[43] Telit. GM862-GPS Hardware User Guide.

[44] C. Troncoso, G. Danezis, E. Kosta, and B. Preneel. PriPAYD: privacy friendly pay-as-you-drive insurance. In Peng Ning and Ting Yu, editors, *Proceedings of the 2007 ACM Workshop on Privacy in the Electronic Society, WPES 2007*, pages 99–107. ACM, 2007.

[45] J. Tsai, S. Egelman, L. Cranor, and A. Acquisti. The effect of online privacy information on purchasing behavior: An experimental study, working paper. In *The 6th Workshop on the Economics of Information Security*, 2007.

# A Security Definition of Optimistic Payment

**Ideal-world/real-world paradigm.** We use the ideal-world/real-world paradigm to prove our construction secure. In this paradigm, parties are modeled as probabilistic polynomial time interactive Turing machines. A protocol $\psi$ is secure if there exists no environment $\mathcal{Z}$ that can distinguish whether it is interacting with adversary $\mathcal{A}$ and parties running protocol $\psi$ or with the ideal process for carrying out the desired task, where ideal adversary $\mathcal{S}$ and dummy parties interact with an ideal functionality $\mathcal{F}_\psi$. More formally, we say that protocol $\psi$ emulates the ideal process if, for any adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for all environments $\mathcal{Z}$, the ensembles $\text{IDEAL}_{\mathcal{F}_\psi,\mathcal{S},\mathcal{Z}}$ and $\text{REAL}_{\psi,\mathcal{A},\mathcal{Z}}$ are computationally indistinguishable. We refer to [11] for a description of these ensembles.

Our construction operates in the $\mathcal{F}_{\text{REG}}$-hybrid model, where parties register their public keys at a trusted registration entity and obtain from it a common reference string. Below we depict the ideal functionality $\mathcal{F}_{\text{REG}}$, which is parameterized with a set of participants $\mathcal{P}$ that is restricted to contain OBU, TSP and TC only. We also describe an ideal functionality $\mathcal{F}_{\text{OP}}$ for Optimistic Payment. Every functionality and every protocol invocation should be instantiated with a unique session-ID that distinguishes it from other instantiations. For the sake of ease of notation, we omit session-IDs from our description.

**Functionality $\mathcal{F}_{\text{REG}}$.** Parameterized with a set of parties $\mathcal{P}$, $\mathcal{F}_{\text{REG}}$ works as follows:

- On input $(\text{crs})$ from party $P$, if $P \notin \mathcal{P}$ it aborts. Otherwise, if there is no value $r$ recorded, it picks $r \leftarrow D$ and records $r$. It sends $(\text{crs}, r)$ to $P$.
- Upon receiving $(\text{register}, v)$ from party $P \in \mathcal{P}$, it records the value $(P, v)$.
- Upon receiving $(\text{retrieve}, P)$ from party $P' \in \mathcal{P}$, if $(P, v)$ is recorded then return $(\text{retrieve}, P, v)$ to $P'$. Otherwise send $(\text{retrieve}, P, \perp)$ to $P'$.

**Functionality $\mathcal{F}_{\text{OP}}$.** Running with OBU, TSP and TC, $\mathcal{F}_{\text{OP}}$ works as follows:

- On input a message $(\text{initialize}, f, \mu)$ from TSP, where $f$ is a mapping $f : (loc, time) \rightarrow \Upsilon$ and $\mu : (\phi, (loc, time)) \rightarrow \{\text{accept}, \text{reject}\}$, $\mathcal{F}_{\text{OP}}$ stores $(f, \mu)$ and sends $(\text{initialize}, f, \mu)$ to OBU.
- On input a message $(\text{payment}, tag, fee, (k, (loc_k, time_k), p_k)_{k=1}^N)$ from OBU, where $tag$ identifies the tax period, $\mathcal{F}_{\text{OP}}$ checks that a message $(\text{payment}, tag, \ldots)$ was not received before, that for $k = 1$ to $N$, $p_k \in \Upsilon$, and that $fee = \sum_{k=1}^N p_k$. If these checks succeed, $\mathcal{F}_{\text{OP}}$ sends $(\text{payment}, tag, fee, N)$ to TSP and stores the tuple $(tag, fee, (k, (loc_k, time_k), p_k)_{k=1}^N)$. Otherwise $\mathcal{F}_{\text{OP}}$ sends $(\text{payment}, tag, \perp)$ and stores $(tag, \perp)$.
- On input a message $(\text{proof}, tag, \phi)$ from TC, $\mathcal{F}_{\text{OP}}$ stores $(tag, \phi)$ and sends $(\text{proof}, tag, \phi)$ to TSP.
- On input a message $(\text{verify}, tag, \phi)$ from TSP, $\mathcal{F}_{\text{OP}}$ checks that it stores messages $(\text{payment}, tag, \ldots)$ and $(\text{proof}, tag, \phi)$. If it is the case, $\mathcal{F}_{\text{OP}}$ sends $(\text{verifyreq}, tag, \phi)$ to OBU. Upon receiving $(\text{verifyresp}, tag, (\sigma, (loc'_\sigma, time'_\sigma), p'_\sigma))$, $\mathcal{F}_{\text{OP}}$ checks whether the stored payment tuple $(k, (loc_k, time_k), p_k)$ equals $(\sigma, (loc'_\sigma, time'_\sigma), p'_\sigma)$ for $k = \sigma$, whether $\mu(\phi, (loc'_\sigma, time'_\sigma))$ outputs accept, and whether $p'_\sigma = f(loc'_\sigma, time'_\sigma)$. If these checks are correct, $\mathcal{F}_{\text{OP}}$ sends $(\text{verifyresul}, \textit{not guilty}, (\sigma, (loc'_\sigma, time'_\sigma), p'_\sigma))$ to TSP. Otherwise it sends $(\text{verifyresul}, \textit{guilty}, (\sigma, (loc'_\sigma, time'_\sigma), p'_\sigma))$.
- On input a message $(\text{blame}, tag)$ from TSP, $\mathcal{F}_{\text{OP}}$ checks that messages $(\text{payment}, tag, \ldots)$, $(\text{proof}, tag, \phi)$ and $(\text{verifyresp}, tag, \ldots)$ were previously received, and in this case it proceeds with the same checks done for $(\text{verify}, \ldots)$. It sends to TC either $(\textit{guilty})$ or $(\textit{not guilty})$.

# B Construction of an Optimistic Payment Scheme

We use several existing results to prove statements about discrete logarithms: (1) proof of knowledge of a discrete logarithm modulo a prime [41]; (2) proof of knowledge of the equality of some element in different representations [12]; (3) proof with interval checks [37] and (4) proof of the disjunction or conjunction of any two of the previous [14]. These results are often given in the form of $\Sigma$-protocols but they can be turned into non-interactive zero-knowledge arguments in the random oracle model via the Fiat-Shamir heuristic [24].

When referring to the proofs above, we follow the notation introduced by Camenisch and Stadler [10] for various proofs of knowledge of discrete logarithms and proofs of the validity of statements about discrete logarithms. $\mathsf{NIPK}\{(\alpha, \beta, \delta) : y = {g_0}^\alpha {g_1}^\beta \wedge \tilde{y} = \tilde{g_0}^\alpha \tilde{g_1}^\delta \wedge A \leq \alpha \leq B\}$ denotes a "*zero-knowledge Proof of Knowledge of integers $\alpha$, $\beta$, and $\delta$ such that $y = {g_0}^\alpha {g_1}^\beta$, $\tilde{y} = \tilde{g_0}^\alpha \tilde{g_1}^\delta$ and $A \leq \alpha \leq B$ holds*", where $y, g_0, g_1, \tilde{y}, \tilde{g_0}$, and $\tilde{g_1}$ are elements of some groups $G = \langle g_0 \rangle = \langle g_1 \rangle$ and $\tilde{G} = \langle \tilde{g_0} \rangle = \langle \tilde{g_1} \rangle$ that have the same order. (Note that some elements in the representation of $y$ and $\tilde{y}$ are equal.) The convention is that letters in the parenthesis, in this example $\alpha$, $\beta$, and $\delta$, denote quantities whose knowledge is being proven, while all other values are known to the verifier. We denote a non-interactive proof of signature possession as $\mathsf{NIPK}\{(x, s_x) : \mathsf{SigVerify}(pk, x, s_x) = \text{accept}\}$.



## B.1 Construction

We begin with a high level description of the optimistic payment scheme. We assume that each party registers its public key at $\mathcal{F}_{\text{REG}}$, and retrieves public keys from other parties by querying $\mathcal{F}_{\text{REG}}$. They also retrieve the common reference string $params_{Com}$, which is computed by algorithm $\mathsf{SetupOP}$.

**Optimistic Payment**

When TSP is activated with $(\text{initialize}, f, \mu)$, TSP runs $\mathsf{TSPkg}(1^k)$ to obtain $(sk_{\text{TSP}}, pk_{\text{TSP}})$, and obtains a setup $params$ with $\mathsf{TSPinit}(f, sk_{\text{TSP}})$. TSP stores $\text{TSP}_0 = (f, \mu, sk_{\text{TSP}}, pk_{\text{TSP}}, params_{Com}, params)$ and sends $(f, \mu, params)$ to OBU. OBU runs $\mathsf{OBUkg}(1^k)$ to get $(sk_{\text{OBU}}, pk_{\text{OBU}})$ and executes $\mathsf{OBUinit}(params, pk_{\text{TSP}})$ to get a bit $b$. If $b = 0$, OBU rejects $params$. Otherwise OBU stores the tuple $\text{OBU}_0 = (f, \mu, sk_{\text{OBU}}, pk_{\text{OBU}}, pk_{\text{TSP}}, params_{Com}, params)$.

When OBU is activated with $(\text{payment}, tag, fee, (k, (loc_k, time_k), p_k)_{k=1}^N)$ and OBU has previously received $(f, \mu, params)$, OBU runs algorithm $\text{Pay}(params_{Com}, params, pk_{\text{OBU}}, sk_{\text{OBU}}, pk_{\text{TSP}}, tag, fee, (k, (loc_k, time_k), p_k)_{k=1}^N)$ to obtain a payment message $m$ along with a signature $s_m$, and auxiliary information $aux$. OBU sets $aux = (aux, (k, (loc_k, time_k), p_k)_{k=1}^N)$, stores $\text{OBU}_{tag} = (\text{OBU}_0, m, s_m, aux)$ and sends $(m, s_m)$ to TSP. TSP runs $\mathsf{VerifyPayment}(params_{Com}, pk_{\text{OBU}}, pk_{\text{TSP}}, m, s_m)$ to obtain a bit $b$. If $b = 0$, TSP rejects $(m, s_m)$. Otherwise TSP stores $\text{TSP}_{tag} = (\text{TSP}_0, m, s_m, pk_{\text{OBU}})$.

When TC is activated with $(\text{proof}, tag, \phi)$, TC runs $\mathsf{TCkg}(1^k)$ to get $(pk_{\text{TC}}, sk_{\text{TC}})$, runs $\mathsf{Prove}(sk_{\text{TC}}, tag, \phi)$ to obtain a proof $Q$ and sends $(Q)$ to TSP. TSP runs $\mathsf{VerifyProof}(pk_{\text{TC}}, Q)$ and aborts if $b = 0$. Otherwise TSP stores $\text{TSP}_{tag} = (\text{TSP}_{tag}, Q)$.

When TSP is activated with $(\text{verify}, tag, \phi)$, and TSP has previously obtained $(m, s_m)$ and $(Q)$, TSP sends $(Q)$ to OBU. OBU executes $\mathsf{VerifyProof}(pk_{\text{TC}}, Q)$ and aborts if $b = 0$. Otherwise OBU runs $\mathsf{OBUopen}(sk_{\text{OBU}}, Q, aux)$ to get a response $R$ and sends $(R)$ to TSP. TSP runs $\mathsf{Check}(params_{Com}, pk_{\text{OBU}}, pk_{\text{TSP}}, m, s_m, Q, R)$ to obtain either $(\textit{not guilty}, (k, (loc_k, time_k), p_k)$ or $(\textit{guilty}, (k, (loc_k, time_k), p_k))$.

When TSP is activated with $(\text{blame}, tag)$, and messages $(m, s_m)$, $(Q)$ and $(R)$ were previously received, TSP sends $((m, s_m), R)$ to TC. TC runs $\mathsf{Check}(params_{Com}, pk_{\text{OBU}}, pk_{\text{TSP}}, m, s_m, Q, R)$ to obtain $(\textit{not guilty}, (k, (loc_k, time_k), p_k))$ or $(\textit{guilty}, (k, (loc_k, time_k), p_k))$.

In the following, we denote the signature algorithms used by TSP, OBU and TC as $(\text{TSPkeygen}, \text{TSPsign}, \text{TSPverify})$, $(\text{OBUkeygen}, \text{OBUsign}, \text{OBUverify})$ and $(\text{TCkeygen}, \text{TCsign}, \text{TCverify})$. $H$ stands for a collision-resistant hash function, which is modeled as a random oracle.

$\mathsf{SetupOP}(1^k)$**.** Run $\mathsf{ComSetup}(1^k)$ and output $params_{Com}$.

$\mathsf{TSPkg}(1^k)$**.** Run $\text{TSPkeygen}(1^k)$ to get a key pair $(pk_{\text{TSP}}, sk_{\text{TSP}})$. Output $(pk_{\text{TSP}}, sk_{\text{TSP}})$.

$\mathsf{OBUkg}(1^k)$**.** Run $\text{OBUkeygen}(1^k)$ to get a key pair $(pk_{\text{OBU}}, sk_{\text{OBU}})$. Output $(pk_{\text{OBU}}, sk_{\text{OBU}})$.

$\mathsf{TCkg}(1^k)$**.** Run $\text{TCkeygen}(1^k)$ to obtain a key pair $(pk_{\text{TC}}, sk_{\text{TC}})$. Output $(pk_{\text{TC}}, sk_{\text{TC}})$.

$\mathsf{TSPinit}(f, sk_{\text{TSP}})$**.** For all possible prices $p \in \Upsilon$, run $s = \text{TSPsign}(sk_{\text{TSP}}, p)$ and output the set $params = (p, s)$.

$\mathsf{OBUinit}(params, pk_{\text{TSP}})$**.** Parse $params$ as $(p, s)$ and run $\mathsf{TSPverify}(pk_{\text{TSP}}, p, s)$ for all $p \in \Upsilon$. If all the signatures are correct, output $b = 1$ else $b = 0$.

$\text{Pay}(params_{Com}, params, pk_{\text{OBU}}, sk_{\text{OBU}}, pk_{\text{TSP}}, tag\ fee, (k, (loc_k, time_k), p_k)_{k=1}^N)$. For $k = 1$ to $N$, execute $h_k = H(loc_k, time_k)$, calculate a commitment to the price $(c_k, open_k) = \mathsf{Commit}(params_{Com}, p_k)$ and compute a proof of possession of a signature on the price $\pi_k = \mathsf{NIPK}\{(p_k, open_k, s_k) : \text{TSPverify}(pk_{\text{TSP}}, p_k, s_k) = \text{accept} \wedge (c_k, open_k) = \mathsf{Commit}(params_{Com}, p_k)\}$. Add all the prices to obtain the total fee $fee$ and all the openings $open_k$ to get an opening $open_{fee}$ to the commitment to the fee. Set payment message $m = (tag, fee, open_{fee}, (h_k, c_k, \pi_k)_{k=1}^N)$ and run $s_m = \text{OBUsign}(sk_{\text{OBU}}, m)$. Output $(m, s_m)$ and $aux = (open_k)_{k=1}^N$.

$\mathsf{VerifyPayment}(params_{Com}, pk_{\text{OBU}}, pk_{\text{TSP}}, m, s_m)$**.** Parse $m$ as $(tag, fee, open_{fee}, (h_k, c_k, \pi_k)_{k=1}^N)$. For $k = 1$ to $N$, verify $\pi_k$. Add all the commitments to obtain a commitment to the total fee $c_{fee}$, and run $\mathsf{Open}(params_{Com}, c_{fee}, fee, open_{fee})$. If the opening is correct, output $b = 1$. Otherwise output $b = 0$.

$\mathsf{Prove}(sk_{\text{TC}}, tag, \phi)$**.** Set $q = (tag, \phi)$ and run $s_q = \text{TCsign}(sk_{\text{TC}}, q)$. Output $Q = (q, s_q)$.

$\mathsf{VerifyProof}(pk_{\text{TC}}, Q)$**.** Parse $Q$ as $(q, s_q)$ and run $\text{TCverify}(pk_{\text{TC}}, q, s_q)$. Output $b = 1$ if the signature is correct and $b = 0$ otherwise.

$\mathsf{OBUopen}(sk_{\text{OBU}}, Q, aux)$**.** Parse proof $Q$ as $(q, s_q)$, $q$ as $(tag, \phi)$ and $aux$ as $(open_k, (k, (loc_k, time_k), p_k))_{k=1}^N$. Find the data structure $(loc_k, time_k)$ such that $\mu(\phi, (loc_k, time_k))$ outputs accept. Set $r = (tag, (k, (loc_k, time_k), p_k), open_k)$ and run $s_r = \text{OBUsign}(sk_{\text{OBU}}, r)$. Output $R = (r, s_r)$.

$\mathsf{Check}(params_{Com}, pk_{\text{OBU}}, pk_{\text{TSP}}, m, s_m, Q, R)$**.** Parse $R$ as $(r, s_r)$ and run $\mathsf{OBUverify}(pk_{\text{OBU}}, r, s_r)$. If the signature is correct, parse $r$ as $(tag, (\sigma, (loc'_\sigma, time'_\sigma), p'_\sigma), open_\sigma)$, $Q$ as $((tag, \phi), s_q)$ and $m$ as $(tag, fee, open_{fee}, (h_k, c_k,$

$\pi_k)_{k=1}^N$). Check that $open_{fee}$ was picked from the adequate interval. Compute $h'_\sigma = H(loc'_\sigma, time'_\sigma)$, check if $h'_\sigma = h_\sigma$ and if $\mu(\phi, (loc'_\sigma, time'_\sigma))$ outputs accept. If it is the case, set $reasonpos = 0$ and otherwise $reasonpos = 1$. Compute $p_\sigma = f(loc'_\sigma, time'_\sigma)$ and check if $p_\sigma = p'_\sigma$. Run Open$(params_{Com}, c_k, p_k, open_k)$. If it opens correctly set $reasonprice = 0$ and otherwise $reasonprice = 1$. If $reasonpos = reasonprice = 0$, output $(not\ guilty, (k, (loc_k, time_k), p_k))$. If not, output $(guilty, (k, (loc_k, time_k), p_k))$.

**Theorem 1** *This OP scheme securely realizes $\mathcal{F}_{OP}$.*

We prove Theorem 1 in the extended version of this work [3].

## B.2 Efficient Instantiation

We propose an efficient instantiation for the commitment scheme, TSP's signature scheme and the non-interactive proof of signature possession that are used in the construction described in the previous section. The signature schemes of TC and OBU can be instantiated with any existentially unforgeable signature scheme.

**Signature Scheme.** We select the signature scheme proposed by Camenisch and Lysyanskaya [9].

- SigKeygen. On input $1^k$, generate two safe primes $p, q$ of length $k$ such that $p = 2p' + 1$ and $q = 2q' + 1$. The special RSA modulus of length $l_n$ is defined as $n = pq$. Output secret key $sk = (p, q)$. Choose uniformly at random $S \in_R QR_n$, and $R, Z \in_R \langle S \rangle$. Output public key $pk = (n, R, S, Z)$.
- SigSign. On input message $x$ of length $l_x$, choose a random prime number $e$ of length $l_e \geq l_x + 3$, and a random number $v$ of length $l_v = l_n + l_x + l_r$, where $l_r$ is a security parameter [9]. Compute the value $A$ such that $Z \equiv A^e R^x S^v (\text{mod } n)$. Output the signature $(A, e, v)$.
- SigVerify. On inputs message $x$ and signature $(A, e, v)$, check that $Z = A^e R^x S^v (\text{mod } n)$ and $2^{l_e} \leq e \leq 2^{l_e - 1}$.

**Commitment Scheme.** We select the integer commitment scheme due to Damgard and Fujisaki [15].

- ComSetup. Given a special RSA modulus, pick a random generator $g_1 \in_R QR_n$. Pick random $\alpha \leftarrow \{0,1\}^{l_n + l_z}$ and compute $g_0 = g_1^\alpha$. Output parameters $(g_0, g_1, n)$.
- Commit. On input message $x$ of length $l_x$, choose a random number $open_x \in \{0,1\}^{l_n + l_z}$, and compute $c_x = {g_0}^x {g_1}^{open_x} (\text{mod } n)$. Output the commitment $c_x$ and the opening $open_x$.
- Open. On inputs message $x$ and opening $open_x$, compute $c'_x = {g_0}^x {g_1}^{open_x} (\text{mod } n)$ and check whether $c_x = c'_x$.

**Non-Interactive Zero-Knowledge Argument.** We employ the proof of possession of a signature in [9]. Given a signature $(A, e, v)$ on message $x$ and a commitment to the message $c_x = {g_0}^x {g_1}^{open_x}$, the prover computes $\tilde{A} = A g^w$, a commitment $c_w = g^w h^{open_w}$ and a proof that:

$$\begin{aligned}\text{NIPK}\{\ & (x, open_x, e, v, w, open_w, w \cdot e, open_w \cdot e) : \\ & c_x = {g_0}^x {g_1}^{open_x} \ \wedge\ Z = \tilde{A}^e R^x S^v (1/g_0)^{w \cdot e} \ \wedge \\ & c_w = {g_0}^w {g_1}^{open_w} \ \wedge\ 1 = c_w^e (1/g_0)^{w \cdot e} \\ & (1/g_1)^{open_w \cdot e} \ \wedge\ e \in \{0,1\}^{l_e + l_c + l_z} \ \wedge \\ & x \in \{0,1\}^{l_x + l_c + l_z}\}\end{aligned}$$

We turn it into a non-interactive zero-knowledge argument via the Fiat-Shamir heuristic. The prover picks random values:

$$\begin{aligned} r_x &\leftarrow \{0,1\}^{l_x + l_c + l_z}, & r_{open_x} &\leftarrow \{0,1\}^{l_n + l_c + l_z} \\ r_w &\leftarrow \{0,1\}^{l_n + l_c + l_z}, & r_{open_w} &\leftarrow \{0,1\}^{l_n + l_c + l_z} \\ r_e &\leftarrow \{0,1\}^{l_e + l_c + l_z}, & r_{w \cdot e} &\leftarrow \{0,1\}^{l_n + l_e + l_c + l_z} \\ r_v &\leftarrow \{0,1\}^{l_v + l_c + l_z}, & r_{open_w \cdot e} &\leftarrow \{0,1\}^{l_n + l_e + l_c + l_z} \end{aligned}$$

and computes commitments:

$$\begin{aligned} t_{c_x} &= {g_0}^{r_x} {g_1}^{r_{open_x}}, \ t_{c_w} = g^{r_w} h^{r_{open_w}} \\ t'_Z &= \tilde{A}^{r_e} R^{r_x} S^{r_v} (1/g_0)^{r_{w \cdot e}}, \\ t' &= c_w^{r_e} (1/g_0)^{r_{w \cdot e}} (1/g_1)^{r_{open_w \cdot e}}. \end{aligned}$$

Let the challenge computed by the prover be:

$$ch = H(n||g_0||g_1||\tilde{A}||R||S||1/g_0||1/g_1||c_x||Z|| c_w||1||t_{c_x}||t_Z||t_{c_w}||t\ ).$$

The prover computes responses:

$$\begin{aligned} s_x &= r_x - ch \cdot x, & s_{open_x} &= r_{open_x} - ch \cdot open_x \\ s_w &= r_w - ch \cdot w, & s_{open_w} &= r_{open_w} - ch \cdot open_w \\ s_e &= r_e - ch \cdot e, & s_{w \cdot e} &= r_{w \cdot e} - ch \cdot (w \cdot e) \\ s_v &= r_v - ch \cdot v, & & \\ s_{open_w \cdot e} &= r_{open_w \cdot e} - ch \cdot (open_w \cdot e) & & \end{aligned}$$

and sends to the verifier:

$$\pi = (\tilde{A}, c_w, ch, s_x, s_{open_x}, s_e, s_v, s_w, s_{open_w}, s_{w \cdot e}, s_{open_w \cdot e})\ .$$

The verifier computes:

$$\begin{aligned} t'_{c_x} &= c_x^{ch} {g_0}^{s_m} {g_1}^{s_{open_x}}, \ t'_{c_w} = c_w^{ch} {g_0}^{s_w} {g_1}^{s_{open_w}} \\ t'_Z &= Z^{ch} \tilde{A}^{s_e} R^{s_x} S^{s_v} (1/g_0)^{s_{w \cdot e}}, \\ t' &= C_w^{s_e} (1/g_0)^{s_{w \cdot e}} (1/g_1)^{s_{open_w \cdot e}} \end{aligned}$$

and checks whether:

$$s_e \in \{0,1\}^{l_e + l_c + l_z}, \quad s_x \in \{0,1\}^{l_x + l_c + l_z}$$

and finally:

$$ch = H(n||g_0||g_1||\tilde{A}||R||S||1/g_0||1/g_1||c_x||Z|| c_w||1||t'_{c_x}||t'_Z||t'_{c_w}||t'\ ).$$



# An Analysis of Private Browsing Modes in Modern Browsers

Gaurav Aggarwal Elie Burzstein Collin Jackson Dan Boneh
*Stanford University* *CMU* *Stanford University*

## Abstract

We study the security and privacy of private browsing modes recently added to all major browsers. We first propose a clean definition of the goals of private browsing and survey its implementation in different browsers. We conduct a measurement study to determine how often it is used and on what categories of sites. Our results suggest that private browsing is used differently from how it is marketed. We then describe an automated technique for testing the security of private browsing modes and report on a few weaknesses found in the Firefox browser. Finally, we show that many popular browser extensions and plugins undermine the security of private browsing. We propose and experiment with a workable policy that lets users safely run extensions in private browsing mode.

## 1 Introduction

The four major browsers (Internet Explorer, Firefox, Chrome and Safari) recently added private browsing modes to their user interfaces. Loosely speaking, these modes have two goals. First and foremost, sites visited while browsing in private mode should leave no trace on the user's computer. A family member who examines the browser's history should find no evidence of sites visited in private mode. More precisely, a *local attacker* who takes control of the machine at time $T$ should learn no information about private browsing actions prior to time $T$. Second, users may want to hide their identity from web sites they visit by, for example, making it difficult for web sites to link the user's activities in private mode to the user's activities in public mode. We refer to this as privacy from a *web attacker*.

While all major browsers support private browsing, there is a great deal of inconsistency in the type of privacy provided by the different browsers. Firefox and Chrome, for example, attempt to protect against a local attacker and take some steps to protect against a web attacker, while Safari only protects against a local attacker. Even within a single browser there are inconsistencies. For example, in Firefox 3.6, cookies set in public mode are not available to the web site while the browser is in private mode. However, passwords and SSL client certificates stored in public mode are available while in private mode. Since web sites can use the password manager as a crude cookie mechanism, the password policy is inconsistent with the cookie policy.

Browser plug-ins and extensions add considerable complexity to private browsing. Even if a browser adequately implements private browsing, an extension can completely undermine its privacy guarantees. In Section 6.1 we show that many widely used extensions undermine the goals of private browsing. For this reason, Google Chrome disables all extensions while in private mode, negatively impacting the user experience. Firefox, in contrast, allows extensions to run in private mode, favoring usability over security.

**Our contribution.** The inconsistencies between the goals and implementation of private browsing suggests that there is considerable room for research on private browsing. We present the following contributions.

- **Threat model.** We begin with a clear definition of the goals of private browsing. Our model has two somewhat orthogonal goals: security against a local attacker (the primary goal of private browsing) and security against a web attacker. We show that correctly implementing private browsing can be non-trivial and in fact all browsers fail in one way or another. We then survey how private browsing is implemented in the four major browsers, highlighting the quirks and differences between the browsers.

- **Experiment.** We conduct an experiment to test how private browsing is used. Our study is based on a technique we discovered to remotely test if a browser is currently in private browsing mode. Using this technique we post ads on ad-networks and

determine how often private mode is used. Using ad targeting by the ad-network we target different categories of sites, enabling us to correlate the use of private browsing with the type of site being visited. We find it to be more popular at adult sites and less popular at gift sites, suggesting that its primary purpose may not be shopping for "surprise gifts." We quantify our findings in Section 4.

- **Tools.** We describe an automated technique for identifying failures in private browsing implementations and use it to discover a few weaknesses in the Firefox browser.

- **Browser extensions.** We propose an improvement to existing approaches to extensions in private browsing mode, preventing extensions from unintentionally leaving traces of the private activity on disk. We implement our proposal as a Firefox extension that imposes this policy on other extensions.

**Organization.** Section 2 presents a threat model for private browsing. Section 3 surveys private browsing mode in modern browsers. Section 4 describes our experimental measurement of private browsing usage. Section 5 describes the weaknesses we found in existing private browsing implementations. Section 6 addresses the challenges introduced by extensions and plug-ins. Section 7 describes additional related work. Section 8 concludes.

## 2 Private browsing: goal and threat model

In defining the goals and threat model for private browsing, we consider two types of attackers: an attacker who controls the user's machine (a local attacker) and an attacker who controls web sites that the user visits (a web attacker). We define security against each attacker in turn. In what follows we refer to the user browsing the web in private browsing mode as *the user* and refer to someone trying to determine information about the user's private browsing actions as *the attacker*.

### 2.1 Local attacker

Stated informally, security against a local attacker means that an attacker who takes control of the machine *after* the user exits private browsing can learn nothing about the user's actions while in private browsing. We define this more precisely below.

We emphasize that the local attacker has no access to the user's machine before the user exits private browsing. Without this limitation, security against a local attacker is impossible; an attacker who has access to the user's machine before or during a private browsing session can simply install a key-logger and record all user actions. By restricting the local attacker to "after the fact" forensics, we can hope to provide security by having the browser adequately erase persistent state changes during a private browsing session.

As we will see, this requirement is far from simple. For one thing, not all state changes during private browsing should be erased at the end of a private browsing session. We draw a distinction between four types of persistent state changes:

1. Changes initiated by a web site without any user interaction. A few examples in this category include setting a cookie, adding an entry to the history file, and adding data to the browser cache.
2. Changes initiated by a web site, but requiring user interaction. Examples include generating a client certificate or adding a password to the password database.
3. Changes initiated by the user. For example, creating a bookmark or downloading a file.
4. Non-user-specific state changes, such as installing a browser patch or updating the phishing block list.

All browsers try to delete state changes in category (1) once a private browsing session is terminated. Failure to do so is treated as a private browsing violation. However, changes in the other three categories are in a gray area and different browsers treat these changes differently and often inconsistently. We discuss implementations in different browsers in the next section.

To keep our discussion general we use the term *protected actions* to refer to state changes that should be erased when leaving private browsing. It is up to each browser vendor to define the set of protected actions.

**Network access.** Another complication in defining private browsing is server side violations of privacy. Consider a web site that inadvertently displays to the world the last login time of every user registered at the site. Even if the user connects to the site while in private mode, the user's actions are open for anyone to see. In other words, web sites can easily violate the goals of private browsing, but this should not be considered a violation of private browsing in the browser. Since we are focusing on browser-side security, our security model defined below ignores server side violations. While browser vendors mostly ignore server side violations, one can envision a number of potential solutions:

- Much like the phishing filter, browsers can consult a block list of sites that should not be accessed while in private browsing mode.
- Alternatively, sites can provide a P3P-like policy statement saying that they will not violate private browsing. While in private mode, the browser will not connect to sites that do not display this policy.
- A non-technical solution is to post a privacy seal at web sites who comply with private browsing. Users can avoid non-compliant sites when browsing privately.

**Security model.** Security is usually defined using two parameters: the attacker's capabilities and the attacker's goals. A local private browsing attacker has the following capabilities:

- The attacker does nothing until the user leaves private browsing mode at which point the attacker gets complete control of the machine. This captures the fact that the attacker is limited to after-the-fact forensics.

  In this paper we focus on persistent state violations, such as those stored on disk; we ignore private state left in memory. Thus, we assume that before the attacker takes over the local machine all volatile memory is cleared (though data on disk, including the hibernation file, is fair game). Our reason for ignoring volatile memory is that erasing all of it when exiting private browsing can be quite difficult and, indeed, no browser does it. We leave it as future work to prevent privacy violations resulting from volatile memory.

- While active, the attacker cannot communicate with network elements that contain information about the user's activities while in private mode (e.g. web sites the user visited, caching proxies, etc.). This captures the fact that we are studying the implementation of browser-side privacy modes, not server-side privacy.

Given these capabilities, the attacker's goal is as follows: for a set $S$ of HTTP requests of the attacker's choosing, determine if the browser issued any of those requests while in private browsing mode. More precisely, the attacker is asked to distinguish a private browsing session where the browser makes one of the requests in $S$ from a private browsing session where the browser does not. If the local attacker cannot achieve this goal then we say that the browser's implementation of private browsing is secure. This will be our working definition throughout the paper. Note that since an HTTP request contains the name of the domain visited this definition implies that the attacker cannot tell if the user visited a particular site (to see why set $S$ to be the set of all possible HTTP requests to the site in question). Moreover, even if by some auxiliary information the attacker knows that the user visited a particular site, the definition implies that the attacker cannot tell what the user did at the site.

An alternate definition, which is much harder to achieve, requires that the browser hide whether private mode was used at all. We will not consider this stronger goal in the paper. Similarly, we do not formalize properties of private browsing in case the user never exits private browsing mode.

**Difficulties.** Browser vendors face a number of challenges in securing private browsing against a local attacker. One set of problems is due to the underlying operating system. We give two examples:

First, when connecting to a remote site the browser must resolve the site's DNS name. Operating systems often cache DNS resolutions in a local DNS cache. A local attacker can examine the DNS cache and the TTL values to learn if and when the user visited a particular site. Thus, to properly implement private browsing, the browser will need to ensure that all DNS queries while in private mode do not affect the system's DNS cache: no entries should be added or removed. A more aggressive solution, supported in Windows 2000 and later, is to flush the entire DNS resolver cache when exiting private browsing. None of the mainstream browsers currently address this issue.

Second, the operating system can swap memory pages to the swap partition on disk which can leave traces of the user's activity. To test this out we performed the following experiment on Ubuntu 9.10 running Firefox 3.5.9:

1. We rebooted the machine to clear RAM and setup and mounted a swap file (zeroed out).
2. Next, we started Firefox, switched to private browsing mode, browsed some websites and exited private mode but kept Firefox running.
3. Once the browser was in public mode, we ran a memory leak program a few times to force memory pages to be swapped out. We then ran `strings` on the swap file and searched for specific words and content of the webpages visited while in private mode.

The experiment showed that the swap file contained some URLs of visited websites, links embedded in those pages and sometimes even the text from a page – enough information to learn about the user's activity in private browsing.

This experiment shows that a full implementation of private browsing will need to prevent browser memory pages from being swapped out. None of the mainstream browsers currently do this.

**Non-solutions.** At first glance it may seem that security against a local attacker can be achieved using virtual machine snapshots. The browser runs on top of a virtual machine monitor (VMM) that takes a snapshot of the

browser state whenever the browser enters private browsing mode. When the user exits private browsing the VMM restores the browser, and possibly other OS data, to its state prior to entering private mode. This architecture is unacceptable to browser vendors for several reasons: first, a browser security update installed during private browsing will be undone when exiting private mode; second, documents manually downloaded and saved to the file system during private mode will be lost when exiting private mode, causing user frustration; and third, manual tweaks to browser settings (e.g. the homepage URL, visibility status of toolbars, and bookmarks) will revert to their earlier settings when exiting private mode. For all these reasons and others, a complete restore of the browser to its state when entering private mode is not the desired behavior. Only browser state that reveals information on sites visited should be deleted.

User profiles provide a lightweight approach to implementing the VM snapshot method described above. User profiles store all browser state associated with a particular user. Firefox, for example, supports multiple user profiles and the user can choose a profile when starting the browser. The browser can make a backup of the user's profile when entering private mode and restore the profile to its earlier state when exiting private mode. This mechanism, however, suffers from all the problems mentioned above.

Rather than a snapshot-and-restore approach, all four major browsers take the approach of not recording certain data while in private mode (e.g. the history file is not updated) and deleting other data when exiting private mode (e.g. cookies). As we will see, some data that should be deleted is not.

## 2.2 Web attacker

Beyond a local attacker, browsers attempt to provide some privacy from web sites. Here the attacker does not control the user's machine, but has control over some visited sites. There are three orthogonal goals that browsers try to achieve to some degree:

- **Goal 1:** A web site cannot link a user visiting in private mode to the same user visiting in public mode. Firefox, Chrome, and IE implement this (partially) by making cookies set in public mode unavailable while in private mode, among other things discussed in the next section. Interestingly, Safari ignores the web attacker model and makes public cookies available in private browsing.

- **Goal 2:** A web site cannot link a user in one private session to the same user in another private session. More precisely, consider the following sequence of visits at a particular site: the user visits in public mode, then enters private mode and visits again, exits private mode and visits again, re-activates private mode and visits again. The site should not be able to link the two private sessions to the same user. Browsers implement this (partially) by deleting cookies set while in private mode, as well as other restrictions discussed in the next section.

- **Goal 3:** A web site should not be able to determine whether the browser is currently in private browsing mode. While this is a desirable goal, all browsers fail to satisfy this; we describe a generic attack in Section 4.

Goals (1) and (2) are quite difficult to achieve. At the very least, the browser's IP address can help web sites link users across private browsing boundaries. Even if we ignore IP addresses, a web site can use various browser features to fingerprint a particular browser and track that browser across privacy boundaries. Mayer [14] describes a number of such features, such as screen resolution, installed plug-ins, timezone, and installed fonts, all available through standard JavaScript objects. The Electronic Frontier Foundation recently built a web site called Panopticlick [6] to demonstrate that most browsers can be uniquely fingerprinted. Their browser fingerprinting technology completely breaks private browsing goals (1) and (2) in all browsers.

Torbutton [29] — a Tor client implemented as a Firefox extension — puts considerable effort into achieving goals (1) and (2). It hides the client's IP address using the Tor network and takes steps to prevent browser fingerprinting. This functionality is achieved at a considerable performance and convenience cost to the user.

# 3 A survey of private browsing in modern browsers

All four majors browsers (Internet Explorer 8, Firefox 3.5, Safari 4, and Google Chrome 5) implement a private browsing mode. This feature is called *InPrivate* in Internet Explorer, *Private Browsing* in Firefox and Safari, and *Incognito* in Chrome.

**User interface.** Figure 1 shows the user interface associated with these modes in each of the browsers. Chrome and Internet Explorer have obvious chrome indicators that the browser is currently in private browsing mode, while the Firefox indicator is more subtle and Safari only displays the mode in a pull down menu. The difference in visual indicators has to do with shoulder surfing: can a casual observer tell if the user is currently browsing privately? Safari takes this issue seriously and provides no visual indicator in the browser chrome, while other browsers do provide a persistent indicator. We expect that hiding the visual indicator causes users who turn on private browsing to forget to turn it off. We give some evidence of this phenomenon in Section 4 where we show that the percentage of users who browse the web in private mode is greater in browsers with subtle visual indicators.

Another fundamental difference between the browsers is how they start private browsing. IE and Chrome spawn a new window while keeping old windows open, thus allowing the user to simultaneously use the two modes. Firefox does not allow mixing the two modes. When entering private mode it hides all open windows and spawns a new private browsing window. Unhiding public windows does nothing since all tabs in these windows are frozen while browsing privately. Safari simply switches the current window to private mode and leaves all tabs unchanged.

**Internal behavior.** To document how the four implementations differ, we tested a variety of browser features that maintain state and observed the browsers' handling of each feature in conjunction with private browsing mode. The results, conducted on Windows 7 using a default browser settings, are summarized in Tables 1, 2 and 3.

*Table 1* studies the types of data set in public mode that are available in private mode. Some browsers block data set in public mode to make it harder for web sites to link the private user to the pubic user (addressing the web attacker model). The Safari column in Table 1 shows that Safari ignores the web attacker model altogether and makes all public data available in private mode except for the web cache. Firefox, IE, and Chrome block access to some public data while allowing access to other data. All three make public history, bookmarks and passwords available in private browsing, but block public cookies and HTML5 local storage. Firefox allows SSL client certs set in public mode to be used in private mode, thus enabling a web site to link the private session to the user's public session. Hence, Firefox's client cert policy is inconsistent with its cookie policy. IE differs from the other three browsers in the policy for form field autocompletion; it allows using data from public mode.

*Table 2* studies the type of data set in private mode that persists after the user leaves private mode. A local attacker can use data that persists to learn user actions in private mode. All four browsers block cookies, history, and HTML5 local storage from propagating to public mode, but persist bookmarks and downloads. Note that all browsers other than Firefox persist server self-signed certificates approved by the user while in private browsing mode. Lewis [35] recently pointed that Chrome 5.0.375.38 persisted the window zoom level for URLs across incognito sessions; this issue has been fixed as of Chrome 5.0.375.53.

*Table 3* studies data that is entered in private mode and persists during that same private mode session. While in private mode, Firefox writes nothing to the history database and similarly no new passwords and no search terms are saved. However, cookies are stored in memory while in private mode and erased when the user exists private mode. These cookies are not written to persistent storage to ensure that if the browser crashes in private mode this data will be erased. The browser's web cache is handled similarly. We note that among the four browsers, only Firefox stores the list of downloaded items in private mode. This list is cleared on leaving private mode.

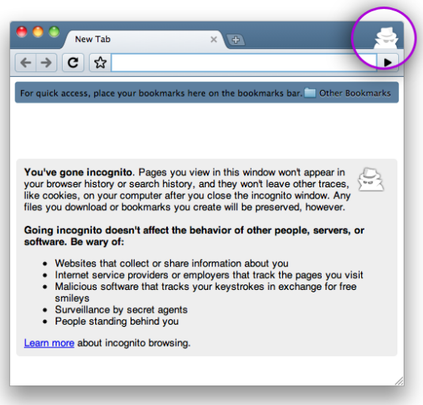
## 3.1 A few initial privacy violation examples

In Section 5.1 we describe tests of private browsing mode that revealed several browser attributes that persist after a private browsing session is terminated. Web sites that use any of these features leave tracks on the user's machine that will enable a local attacker to determine the user's activities in private mode. We give a few examples below.

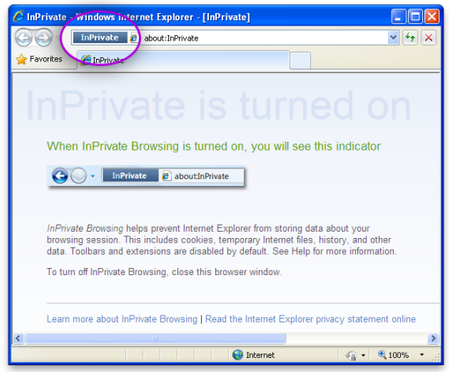
**Custom Handler Protocol.** Firefox implements an HTML 5 feature called *custom protocol handlers* (CPH) that enables a web site to define custom protocols, namely URLs of the form `xyz://site/path` where `xyz` is a custom protocol name. We discovered that custom protocol handlers defined while the browser is in private mode persist after private browsing ends. Consequently, sites that use this feature will leak the fact that the user visited these sites to a local attacker.

**Client Certificate.** IE, Firefox, and Safari support SSL client certificates. A web site can, using JavaScript, instruct the browser to generate an SSL client public/private key pair. We discovered that all these browsers retain the generated key pair even after private browsing ends. Again, if the user visits a site that generates an SSL client key pair, the resulting keys will leak the site's identity to the local attacker. When Internet Explorer and Safari encounter a self-signed certificate they store it in a Microsoft certificate vault. We discovered that entries added to the vault while in private mode remain in the vault when the private session ends. Hence, if the user visits a site that is using a self signed certificate, that information will be available to the local attacker even after the user leaves private mode.

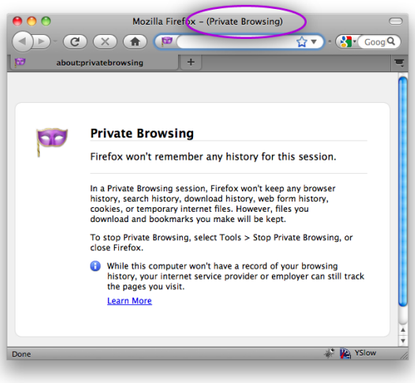
**SMB Query.** Since Internet Explorer shares some underlying components with *Window Explorer* it understands SMB naming conventions such as `\\host\mydir\myfile` and allows the user to browse files and directories. This feature has been used before to steal user data [16]. Here we point out that SMB can also be



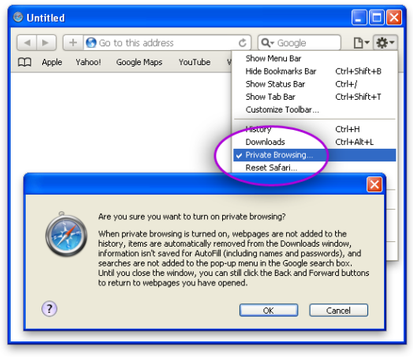

(a) Google Chrome 4




(b) Internet Explorer 8




(c) Firefox 3.6




(d) Safari 4

Figure 1: Private browsing indicators in major browsers

| | FF | Safari | Chrome | IE |
|---|---|---|---|---|
| History | no | yes | no | no |
| Cookies | no | yes | no | no |
| HTML5 local storage | no | yes | no | no |
| Bookmarks | yes | yes | yes | yes |
| Password database | yes | yes | yes | yes |
| Form autocompletion | yes | yes | yes | no |
| User approved SSL self-signed cert | yes | yes | yes | yes |
| Downloaded items list | no | yes | yes | n/a |
| Downloaded items | yes | yes | yes | yes |
| Search box search terms | yes | yes | yes | yes |
| Browser's web cache | no | no | no | no |
| Client certs | yes | yes | yes | yes |
| Custom protocol handlers | yes | n/a | n/a | n/a |
| Per-site zoom level | no | n/a | yes | n/a |

Table 1: Is the state set in earlier public mode(s) accessible in private mode?

| | FF | Safari | Chrome | IE |
|---|---|---|---|---|
| History | no | no | no | no |
| Cookies | no | no | no | no |
| HTML5 Local storage | no | no | no | no |
| Bookmarks | yes | yes | yes | yes |
| Password database | no | no | no | no |
| Form autocompletion | no | no | no | no |
| User approved SSL self-signed cert | no | yes | yes | yes |
| Downloaded items list | no | no | no | n/a |
| Downloaded items | yes | yes | yes | yes |
| Search box search terms | no | no | no | no |
| Browser's web cache | no | no | no | no |
| Client certs | yes | n/a | n/a | yes |
| Custom protocol handlers | yes | n/a | n/a | n/a |
| Per-site zoom level | no | n/a | no | n/a |

Table 2: Is the state set in earlier private mode(s) accessible in public mode?

| | FF | Safari | Chrome | IE |
|---|---|---|---|---|
| History | no | no | no | no |
| Cookies | yes | yes | yes | yes |
| HTML5 Local storage | yes | yes | yes | yes |
| Bookmarks | yes | yes | yes | yes |
| Password database | no | no | no | no |
| Form autocompletion | no | no | no | no |
| User approved SSL self-signed cert | yes | yes | yes | yes |
| Downloaded items list | yes | no | no | n/a |
| Downloaded items | yes | yes | yes | yes |
| Search box search terms | no | no | no | no |
| Browser's web cache | yes | yes | yes | yes |
| Client certs | yes | n/a | n/a | yes |
| Custom protocol handlers | yes | n/a | n/a | n/a |
| Per-site zoom level | no | n/a | yes | n/a |

Table 3: Is the state set in private mode at some point accessible later in the same session?

used to undo some of the benefits of private browsing mode. Consider the following code :

```
<img src="\\[WEB SERVER IP]\image.jpg">
```

When IE renders this tag, it initiates an SMB request to the web server whose IP is specified in the image source. Part of the SMB request is an NTLM authentication that works as follows: first an anonymous connection is tried and if it fails IE starts a challenge-response negotiation. IE also sends to the server *Windows username, Windows domain name, Windows computer name* even when the browser is in InPrivate mode. Even if the user is behind a proxy, clears the browser state, and uses InPrivate, SMB connections identify the user to the remote site. While experimenting with this we found that many ISPs filter the SMB port 445 which makes this attack difficult in practice.

## 4 Usage measurement

We conducted an experiment to determine how the choice of browser and the type of site being browsed affects whether users enable private browsing mode. We used advertisement networks as a delivery mechanism for our measurement code, using the same ad network and technique previously demonstrated in [10, 4].

**Design.** We ran two simultaneous one-day campaigns: a campaign that targeted adult sites, and a campaign that targeted gift shopping sites. We also ran a campaign on news sites as a control. We spent $120 to purchase 155,216 impressions, split evenly as possible between the campaigns. Our advertisement detected private browsing mode by visiting a unique URL in an `<iframe>` and using JavaScript to check whether a link to that URL was displayed as purple (visited) or blue (unvisited). The technique used to read the link color varies by browser; on Firefox, we used the following code:

```
if (getComputedStyle(link).color ==
        "rgb(51,102,160)")
  // Link is purple, private browsing is OFF
} else {
  // Link is blue, private browsing is ON
}
```

To see why this browser history sniffing technique [11] reveals private browsing status, recall that in private mode all browsers do not add entries to the history database. Consequently, they will color the unique URL link as unvisited. However, in public mode the unique URL will be added to the history database and the browser will render the link as visited. Thus, by reading the link color we learn the browser's privacy state. We developed a demonstration of this technique in February 2009 [9]. To the best of our knowledge, we are the first to demonstrate this technique to detect private browsing mode in all major browsers.

While this method correctly detects all browsers in private browsing, it can slightly over count due to false positives. For example, some people may disable the history feature in their browser altogether, which will incorrectly make us think they are in private mode. In Firefox, users can disable the `:visited` pseudotag using a Firefox preference used as a defense against history sniffing. Again, this will make us think they are in private mode. We excluded beta versions of Firefox 3.7 and Chrome 6 from our experiment, since these browsers have experimental visited link defenses that prevent our automated experiment from working. However, we note that these defenses are not sufficient to prevent web attackers from detecting private browsing, since they are not designed to be robust against attacks that involve user interaction [3]. We also note that the experiment only measures the presence of private mode, not the intent of private mode—some users may be in private mode without realizing it.

**Results.** The results of our ad network experiment are shown in Figure 2. We found that private browsing was more popular at adult web sites than at gift shopping sites and news sites, which shared a roughly equal level of private browsing use. This observation suggests that some browser vendors may be mischaracterizing the primary use of the feature when they describe it as a tool for buying surprise gifts [8, 17].

We also found that private browsing was more commonly used in browsers that displayed subtle private browsing indicators. Safari and Firefox have subtle indicators and enforce a single mode across all windows; they had the highest rate of private browsing use. Google Chrome and Internet Explorer give users a separate window for private browsing, and have more obvious private browsing indicators; these browsers had lower rates of private browsing use. These observations suggest that users may remain in private browsing mode for longer if they are not reminded of its existence by a separate window with obvious indicators.

**Ethics.** The experimental design complied with the terms of service of the advertisement network. The servers logged only information that is typically logged by advertisers when their advertisements are displayed. We also chose not to log the client's IP address. We discussed the experiment with the institutional review boards at our respective institutions and were instructed that a formal IRB review was not required because the advertisement did not interact or intervene with individuals or obtain identifiable private information.



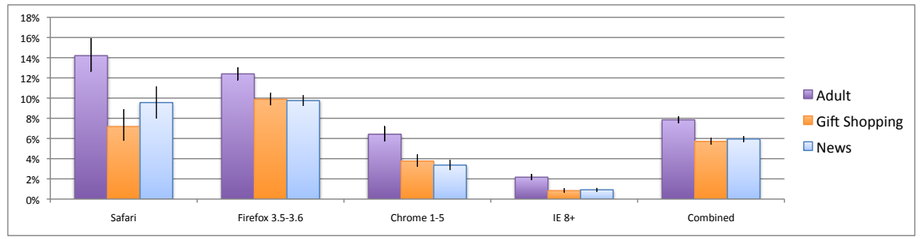



Figure 2: Observed rates of private browsing use

## 5 Weaknesses in current implementations: a systematic study

Given the complexity of modern browsers, a systematic method is needed for testing that private browsing modes adequately defend against the threat models of Section 2. During our blackbox testing in Section 3.1 it became clear that we need a more comprehensive way to ensure that *all* browser features behave correctly in private mode. We performed two systematic studies:

- Our first study is based on a manual review of the Firefox source code. We located all points in the code where Firefox writes to persistent storage and manually verified that those writes are disabled in private browsing mode.
- Our second study is an automated tool that runs the Firefox unit tests in private browsing mode and looks for changes in persistent storage. This tool can be used as a regression test to ensure that new browser features are consistent with private browsing.

We report our results in the next two sections.

### 5.1 A systematic study by manual code review

Firefox keeps all the state related to the user's browsing activity including preferences, history, cookies, text entered in forms fields, search queries, etc. in a *Profile* folder on disk [22]. By observing how and when persistent modifications to these files occur in private mode we can learn a great deal about how private mode is implemented in Firefox. In this section we describe the results of our manual code review of all points in the Firefox code that modify files in the *Profile* folder.

Our first step was to identify those files in the profile folder that contain information about a private browsing session. Then, we located the modules in the Mozilla code base that directly or indirectly modify these files. Finally, we reviewed these modules to see if they write to disk while in private mode.

Our task was greatly simplified by the fact that all writes to files inside the *Profile* directory are done using two code abstractions. The first is `nsIFile`, a cross-platform representation of a location in the filesystem used to read or write to files [21]. The second is `Storage`, a SQLite database API that can be used by other Firefox components and extensions to manipulate SQLite database files [23]. Points in the code that call these abstractions can check the current private browsing state by calling or hooking into the `nsIPrivateBrowsingService` interface [24].

Using this method we located 24 points in the Firefox 3.6 code base that control all writes to sensitive files in the *Profile* folder. Most had adequate checks for private browsing mode, but some did not. We give a few examples of points in the code that do not adequately check private browsing state.

- Security certificate settings (stored in file `cert8.db`): stores all security certificate settings and any SSL certificates that have been imported into Firefox either by an authorized website or manually by the user. This includes SSL client certificates.

  There are no checks for private mode in the code. We explained in Section 3.1 that this is a violation of the private browsing security model since a local attacker can easily determine if the user visited a site that generates a client key pair or installs a client certificate in the browser. We also note that certificates created outside private mode are usable in private mode, enabling a web attacker to link the user in public mode to the same user in private mode.

- Site-specific preferences (stored in file `permissions.sqlite`): stores many of Firefox permissions that are decided on a per-site basis. For example, it stores which sites are allowed or blocked from setting cookies, installing

extensions, showing images, displaying popups, etc.

While there are checks for private mode in the code, not all state changes are blocked. Permissions added to block cookies, popups or allow add-ons in private mode are persisted to disk. Consequently, if a user visits some site that attempts to open a popup, the popup blocker in Firefox blocks it and displays a message with some actions that can be taken. In private mode, the "Edit popup blocker preferences" option is enabled and users who click on that option can easily add a permanent exception for the site without realizing that it would leave a trace of their private browsing session on disk. When browsing privately to a site that uses popups, users might be tempted to add the exception, thus leaking information to the local attacker.

- Download actions (stored in file `mimeTypes.rdf`): the file stores the user's preferences with respect to what Firefox does when it comes across known file types like pdf or avi. It also stores information about which protocol handlers (desktop-based or custom protocol handlers) to launch when it encounters a non-http protocol like `mailto` [26].

  There are no checks for private mode in the code. As a result, a webpage can install a custom protocol handler into the browser (with the user's permission) and this information would be persisted to disk even in private mode. As explained in Section 3.1, this enables a local attacker to learn that the user visited the website that installed the custom protocol handler in private mode.

## 5.2 An automated private browsing test using unit tests

All major browsers have a collection of unit tests for testing browser features before a release. We automate the testing of private browsing mode by leveraging these tests to trigger many browser features that can potentially violate private browsing. We explain our approach as it applies to the Firefox browser. We use MozMill, a Firefox user-interface test automation tool [20]. Mozilla provides about 196 MozMill tests for the Firefox browser.

**Our approach.** We start by creating a fresh browser profile and set preferences to always start Firefox in private browsing mode. Next we create a backup copy of the profile folder and start the MozMill tests. We use two methods to monitor which files are modified by the browser during the tests:

- `fs_usage` is a Mac OSX utility that presents system calls pertaining to filesystem activity. It outputs the name of the system call used to access the filesystem and the file descriptor being acted upon. We built a wrapper script around this tool to map the file descriptors to actual pathnames using `lsof`. We run our script in parallel with the browser and the script monitors all files that the browser writes to.

- We also use the "last modified time" for files in the profile directory to identity those files that are changed during the test.

Once the MozMill test completes we compare the modified profile files with their backup versions and examine the exact changes to eliminate false positives. In our experiments we took care to exclude all MozMill tests like "testPrivateBrowsing" that can turn off private browsing mode. This ensured that the browser was in private mode throughout the duration of the tests.

We did the above experiment on Mac OSX 10.6.2 and Windows Vista running Firefox 3.6. Since we only consider the state of browser profile and start with a clean profile, the results should not depend on OS or state of the machine at the time of running the tests.

**Results.** After running the MozMill tests we discovered several additional browser features that leak information about private mode. We give a few examples.

- Certificate Authority (CA) Certificates (stored in `cert8.db`). Whenever the browser receives a certificate chain from the server, it stores all the certificate authorities in the chain in `cert8.db`. Our tests revealed that CA certs cached in private mode persist when private mode ends. This is significant privacy violation. Whenever the user visits a site that uses a non-standard CA, such as certain government sites, the browser will cache the corresponding CA cert and expose this information to the local attacker.

- SQLite databases. The tests showed that the last modified timestamps of many SQLite databases in the profile folder are updated during the test. But at the end of the tests, the resulting files have exactly the same size and there are no updates to any of the tables. Nevertheless, this behavior can exploited by a local attacker to discover that private mode was turned on in the last browsing session. The attacker simply observes that no entries were added to the history database, but the SQLite databases were accessed.

- Search Plugins (stored in `search.sqlite` and `search.json`). Firefox supports auto-discovery of search plugins [19, 25] which is a way for web sites to advertise their Firefox search plugins to the user. The tests showed that a search plugin added in private mode persists to disk. Consequently, a local attacker will discover that the user visited the web site that provided the search plugin.

- Plugin Registration (stored in `pluginreg.dat`). This file is generated automatically and records information about installed plugins like Flash and Quicktime. We observed changes in modification time, but there were only cosmetic changes in the file content. However, as with search plugins, new plugins installed in private mode result in new information written to `pluginreg.dat`.

Discovering these leaks using MozMill tests is much easier than a manual code review.

**Using our approach as a regression tool.** Using existing unit tests provides a quick and easy way to test private browsing behavior. However, it would be better to include testcases that are designed specifically for private mode and cover all browser components that could potentially write to disk. The same suite of testcases could be used to test all browsers and hence would bring some consistency in the behavior of various browsers in private mode.
As a proof of concept, we wrote two MozMill testcases for the violations discovered in Section 5.1:

- Site-specific Preferences (stored in file `permissions.sqlite`): visits a fixed URL that open up a popup. The test edits preferences to allow a popup from this site.
- Download Actions (`mimeTypes.rdf`): visits a fixed URL that installs a custom protocol handler.

Running these tests using our testing script revealed writes to both profile files involved.

# 6 Browser addons

Browser addons (extensions and plug-ins) pose a privacy risk to private browsing because they can persist state to disk about a user's behavior in private mode. The developers of these addons may not have considered private browsing mode while designing their software, and their source code is not subject to the same rigorous scrutiny that browsers are subjected to. Each of the different browsers we surveyed had a different approach to addons in private browsing mode:

- **Internet Explorer** has a configurable "Disable Toolbars and Extensions when InPrivate Browsing Mode Starts" menu option, which is checked by default. When checked, extensions (browser helper objects) are disabled, although plugins (ActiveX controls) are still functional.

- **Firefox** allows extensions and plugins to function normally in Private Browsing mode.

- **Google Chrome** disables most extension functionality in Incognito mode. However, plugins (including plugins that are bundled with extensions) are enabled. Users can add exceptions on a per-extension basis using the extensions management interface.

- **Safari** does not have a supported extension API. Using unsupported APIs, it is possible for extensions to run in private browsing mode.

In Section 6.1, we discuss problems that can occur in browsers that allow extensions in private browsing mode. In Section 6.2 we discuss approaches to address these problems, and we implement a mitigation in Section 6.3.

## 6.1 Extensions violating private browsing

We conducted a survey of extensions to find out if they violated private browsing mode. This section describes our findings.

**Firefox.** We surveyed the top 40 most popular add-ons listed at `http://addons.mozilla.org`. Some of these extensions like "Cooliris" contain binary components (native code). Since these binary components execute with the same permissions as those of the user, the extensions can, in principle, read or write to any file on disk. This arbitrary behavior makes the extensions difficult to analyze for private mode violations. We regard all *binary extensions* as unsafe for private browsing and focus our attention only on *JavaScript-only extensions*.

To analyze the behavior of JavaScript-only extensions, we observed all persistent writes they caused when the browser is running in private mode. Specifically, for each extension, we install that extension and remove all other extensions. Then, we run the browser for some time, do some activity like visiting websites and modifying extension options so as to exercise as many features of the extension as possible and track all writes that happen during this browsing session. A manual scan of the files and data that were written then tells us if the extension violated private mode. If we find any violations, the extension is *unsafe* for private browsing. Otherwise, it may or may not be safe.

Tracking all writes caused by extensions is easy as almost all JavaScript-only extensions rely on either of the following three abstractions to persist data on disk:

- `nsIFile` is a cross-platform representation of a location in the filesystem. It can be used

to create or remove files/directories and write data when used in combination with components such as `nsIFileOutputStream` and `nsISafeOutputStream`.

- `Storage` is a SQLite database API [23] and can be used to create, remove, open or add new entries to SQLite databases using components like `mozIStorageService`, `mozIStorageStatement` and `mozIStorageConnection`.

- `Preferences` can be used to store preferences containing key-value (boolean, string or integer) pairs using components like `nsIPrefService`, `nsIPrefBranch` and `nsIPrefBranch2`.

We instrumented Firefox (version 3.6 alpha1 pre, codenamed *Minefield*) by adding log statements in all functions in the above Mozilla components that could write data to disk. This survey was done on a Windows Vista machine.

Out of the 32 JavaScript-only extensions, we did not find any violations for 16 extensions. Some of these extensions like "Google Shortcuts" did not write any data at all and some others like "Firebug" only wrote boolean preferences. Other extensions like "1-Click YouTube Video Download" only write files that users want to download whereas "FastestFox" writes bookmarks made by the user. Notably, only one extension ("Tab Mix Plus") checks for private browsing mode and disables the UI option to save session if it is detected.

For 16 extensions, we observed writes to disk that can allow an attacker to learn about private browsing activity. We provide three categories of the most common violations below:

- **URL whitelist/blocklist/queues.** Many extensions maintain a list of special URLs that are always excluded from processing. For instance, "NoScript" extension blocks all scripts running on visited webpages. User can add sites to a whitelist for which it should allow all scripts to function normally. Such exceptions added in private mode are persisted to disk. Also, downloaders like "DownThemAll" maintain a queue of URLs to download from. This queue is persisted to disk even in private mode and not cleared until download completes.

- **URL Mappings.** Some extensions allow specific features or processing to be enabled for specific websites. For instance, "Stylish" allows different CSS styles to be used for rendering pages from different domains. The mapping of which style to use for which website is persisted to disk even in private mode.

- **Timestamp.** Some extensions store a timestamp indicating the last use of some feature or resource. For instance, "Personas" are easy-to-use themes that let the user personalize the look of the browser. It stores a timestamp indicating the last time when the theme was changed. This could potentially be used by an attacker to learn that private mode was turned on by comparing this timestamp with the last timestamp when a new entry was added to the browser history.

It is also interesting to note that the majority of the extensions use `Preferences` or `nsIFile` to store their data and very few use the SQLite database. Out of the 32 JavaScript-only extensions, only two use the SQLite database whereas the rest of them use the former.

**Google Chrome.** Google launched an extension platform for Google Chrome [5] at the end of January 2010. We have begun a preliminary analysis of the most popular extensions that have been submitted to the official extensions gallery. Of the top 100 extensions, we observed that 71 stored data to disk using the `localStorage` API. We also observed that 5 included plugins that can run arbitrary native code, and 4 used Google Analytics to store information about user behavior on a remote server. The significant use of local storage by these extensions suggests that they may pose a risk to Incognito.

### 6.2 Running extensions in private browsing

Current browsers force the user to choose between running extensions in private browsing mode or blocking them. Because not all extensions respect private browsing mode equally, these policies will either lead to privacy problems or block extensions unnecessarily. We recommend that browser vendors provide APIs that enable extension authors to decide which state should be persisted during private browsing and which state should be cleared. There are several reasonable approaches that achieve this goal:

- **Manual check.** Extensions that opt-in to running in private browsing mode can detect the current mode and decide whether or not to persist state.

- **Disallow writes.** Prevent extensions from changing any local state while in private browsing mode.

- **Override option.** Discard changes made by extensions to local state while in private browsing mode, unless the extension explicitly indicates that the write should persist beyond private browsing mode.



Several of these approaches have been under discussion on the Google Chrome developers mailing list [28]. We describe our implementation of the first variant in Section 6.3. We leave the implementation of the latter variants for future work.

### 6.3 Extension blocking tool

To implement the policy of blocking extensions from running in private mode as described in section 6.2, we built a Firefox extension called *ExtensionBlocker* in 371 lines of JavaScript. Its basic functionality is to disable all extensions that are not *safe* for private mode. So, all unsafe extensions will be disabled when the user enters private mode and then re-enabled when the user leaves private mode. An extension is considered safe for private mode if its manifest file (`install.rdf` for Firefox extensions) contains a new XML tag `<privateModeCompatible/>`. Table 4 shows a portion of the manifest file of ExtensionBlocker declaring that it is safe for private browsing.

ExtensionBlocker subscribes to the `nsIPrivateBrowsingService` to observe transitions into and out of private mode. Whenever private mode is enabled, it looks at each enabled extension in turn, checks their manifest file for the `<privateModeCompatible/>` tag and disables the extension if no tag is found. Also, it saves the list of extensions that were enabled before going to private mode. Lastly, when the user switches out of private mode, it re-enables all extensions in this saved list. At this point, it also cleans up the saved list and any other state to make sure that we do not leave any traces behind.

One implementation detail to note here is that we need to restart Firefox to make sure that appropriate extensions are completely enabled or disabled. This means that the browser would be restarted at every entry into or exit from private mode. However, the public browsing session will still be restored after coming out of private mode.

## 7 Related work

**Web attacker.** Most work on private browsing focuses on security against a web attacker who controls a number of web sites and is trying to determine the user's browsing behavior at those sites. Torbutton [29] and FoxTor [31] are two Firefox extensions designed to make it harder for web sites to link users across sessions. Both rely on the Tor network for hiding the client's IP address from the web site. PWS [32] is a related Firefox extension designed for search query privacy, namely preventing a search engine from linking a sequence of queries to a specific user.

Earlier work on private browsing such as [34] focused primarily on hiding the client's IP address. Browser fingerprinting techniques [1, 14, 6] showed that additional steps are needed to prevent linking at the web site. Torbutton [29] is designed to mitigate these attacks by blocking various browser features used for fingerprinting the browser.

Other work on privacy against a web attacker includes Janus [7], Doppelganger [33] and Bugnosis [2]. Janus is an anonymity proxy that also provides the user with anonymous credentials for logging into sites. Doppelganger [33] is a client-side tool that focuses on cookie privacy. The tool dynamically decides which cookies are needed for functionality and blocks all other cookies. Bugnosis [2] is a Firefox extension that warns users about server-side tracking using web bugs. Millet et al. carry out a study of cookie policies in browsers [18].

P3P is a language for web sites to specify privacy policies. Some browsers let users configure the type of sites they are willing to interact with. While much work went into improving P3P semantics [13, 27, 30] the P3P mechanism has not received widespread adoption.

**Local attacker.** In recent years computer forensics experts developed an array of tools designed to process the browser's cache and history file in an attempt to learn what sites a user visited before the machine was confiscated [12]. *Web historian*, for example, will crawl browser activity files and report on all recent activity done using the browser. The tool supports all major browsers. The Forensic Tool Kit (FTK) has similar functionality and an elegant user interface for exploring the user's browsing history. A well designed private browsing mode should successfully hide the user's activity from these tools.

In an early analysis of private browsing modes, McKinley [15] points out that the Flash Player and Google Gears browser plugins violate private browsing modes. Flash player has since been updated to be consistent with the browser's privacy mode. More generally, NPAPI, the plugin API, was extended to allow plugins to query the browser's private browsing settings so that plugins can modify their behavior when private browsing is turned on. We showed that the problem is more complex for browser extensions and proposed ways to identify and block problematic extensions.

## 8 Conclusions

We analyzed private browsing modes in modern browsers and discussed their success at achieving the desired security goals. Our manual review and automated testing tool pointed out several weaknesses in existing

```
<em:targetApplication>
  <Description>
    <em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id>
    <em:minVersion>1.5</em:minVersion>
    <em:maxVersion>3.*</em:maxVersion>
    <em:privateModeCompatible/>
  </Description>
</em:targetApplication>
```

Table 4: A portion of the manifest file of ExtensionBlocker

implementations. The most severe violations enable a local attacker to completely defeat the benefits of private mode. In addition, we performed the first measurement study of private browsing usage in different browsers and on different sites. Finally, we examined the difficult issues of keeping browser extensions and plug-ins from undoing the goals of private browsing.

**Future work.** Our results suggest that current private browsing implementations provide privacy against some local and web attackers, but can be defeated by determined attackers. Further research is needed to design stronger privacy guarantees without degrading the user experience. For example, we ignored privacy leakage through volatile memory. Is there a better browser architecture that can detect all relevant private data, both in memory and on disk, and erase it upon leaving private mode? Moreover, the impact of browser extensions and plug-ins on private browsing raises interesting open problems. How do we prevent uncooperative and legacy browser extensions from violating privacy? In browsers like IE and Chrome that permit public and private windows to exist in parallel, how do we ensure that extensions will not accidentally transfer data from one window to the other? We hope this paper will motivate further research on these topics.

## Acknowledgments

We thank Martin Abadi, Jeremiah Grossman, Sid Stamm, and the USENIX Program Committee for helpful comments about this work. This work was supported by NSF.

## References

[1] 0x000000. Total recall on Firefox. http://mandark.fr/0x000000/articles/Total_Recall_On_Firefox..html.

[2] Adil Alsaid and David Martin. Detecting web bugs with Bugnosis: Privacy advocacy through education. In *Proc. of the 2002 Workshop on Privacy Enhancing Technologies (PETS)*, 2002.

[3] David Baron et al. :visited support allows queries into global history, 2002. https://bugzilla.mozilla.org/show_bug.cgi?id=147777.

[4] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. of the 15th ACM Conference on Computer and Communications Security. (CCS)*, 2008.

[5] Nick Baum. Over 1,500 new features for Google Chrome, January 2010. http://chrome.blogspot.com/2010/01/over-1500-new-features-for-google.html.

[6] Peter Eckersley. A primer on information theory and privacy, January 2010. https://www.eff.org/deeplinks/2010/01/primer-information-theory-and-privacy.

[7] E. Gabber, P. B. Gibbons, Y. Matias, and A. Mayer. How to make personalized web browing simple, secure, and anonymous. In *Proceedings of Financial Cryptography'97*, volume 1318 of *LNCS*, 1997.

[8] Google. Explore Google Chrome features: Incognito mode (private browsing). http://www.google.com/support/chrome/bin/answer.py?hl=en&answer=95464.

[9] Jeremiah Grossman and Collin Jackson. Detecting Incognito, Feb 2009. http://crypto.stanford.edu/~collinj/research/incognito/.

[10] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from DNS rebinding attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[11] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting browser state from web privacy attacks. In *Proc. of the 15th International World Wide Web Conference (WWW)*, 2006.

[12] Keith Jones and Rohyt Belani. Web browser forensics, 2005. www.securityfocus.com/infocus/1827.

[13] Stephen Levy and Carl Gutwin. Improving understanding of website privacy policies with fine-grained policy anchors. In *Proc. of WWW'05*, pages 480–488, 2005.

[14] Jonathan R. Mayer. *"Any person... a pamphleteer": Internet Anonymity in the Age of Web 2.0*. PhD thesis, Princeton University, 2009.

[15] Katherine McKinley. Cleaning up after cookies, Dec. 2008. https://www.isecpartners.com/files/iSEC_Cleaning_Up_After_Cookies.pdf.

[16] Jorge Medina. Abusing insecure features of internet explorer, Febuary 2010. http://www.blackhat.com/presentations/bh-dc-10/Medina_Jorge/BlackHat-DC-2010-Medina-Abusing-/insecure-features-of-Internet-/Explorer-wp.pdf.

[17] Microsoft. InPrivate browsing. http://www.microsoft.com/windows/internet-explorer/features/safer.aspx.

[18] Lynette Millett, Batya Friedman, and Edward Felten. Cookies and web browser design: Toward realizing informed consent online. In *Proce. of the CHI 2001*, pages 46–52, 2001.

[19] Mozilla Firefox - Creating OpenSearch plugins for Firefox. https://developer.mozilla.org/en/Creating_OpenSearch_plugins_for_Firefox.

[20] Mozilla Firefox - MozMill. http://quality.mozilla.org/projects/mozmill.

[21] Mozilla Firefox - nsIFile. https://developer.mozilla.org/en/nsIFile.

[22] Mozilla Firefox - Profiles. http://support.mozilla.com/en-US/kb/Profiles.

[23] Mozilla Firefox - Storage. https://developer.mozilla.org/en/Storage.

[24] Mozilla Firefox - Supporting private browsing mode. https://developer.mozilla.org/En/Supporting_private_browsing_mode.

[25] OpenSearch. http://www.opensearch.org.

[26] Web-based protocol handlers. https://developer.mozilla.org/en/Web-based_protocol_handlers.

[27] The platform for privacy preferences project (P3P). http://www.w3.org/TR/P3P.

[28] Matt Perry. RFC: Extensions Incognito, January 2010. http://groups.google.com/group/chromium-dev/browse_thread/thread/5b95695a7fdf6c15/b4052bb405f2820f.

[29] Mike Perry. Torbutton. http://www.torproject.org/torbutton/design.

[30] J. Reagle and L. Cranor. The platform for privacy preferences. *CACM*, 42(2):48–55, 1999.

[31] Sasha Romanosky. FoxTor: helping protect your identity while browsing online. cups.cs.cmu.edu/foxtor.

[32] F. Saint-Jean, A. Johnson, D. Boneh, and J. Feigenbaum. Private web search. In *Proc. of the 6th ACM Workshop on Privacy in the Electronic Society (WPES)*, 2007.

[33] Umesh Shankar and Chris Karlof. Doppelganger: Better browser privacy without the bother. In *Proceedings of ACM CCS'06*, pages 154–167, 2006.

[34] Paul Syverson, Michael Reed, and David Goldschlag. Private web browsing. *Journal of Computer Security (JCS)*, 5(3):237–248, 1997.

[35] Lewis Thompson. Chrome incognito tracks visited sites, 2010. www.lewiz.org/2010/05/chrome-incognito-tracks-/visited-sites.html.

# BotGrep: Finding P2P Bots with Structured Graph Analysis

Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, Nikita Borisov
University of Illinois at Urbana-Champaign
{sn275,mittal2,hong78,caesar,nikita}@illinois.edu

## Abstract

A key feature that distinguishes modern botnets from earlier counterparts is their increasing use of structured overlay topologies. This lets them carry out sophisticated coordinated activities while being resilient to churn, but it can also be used as a point of detection. In this work, we devise techniques to localize botnet members based on the unique communication patterns arising from their overlay topologies used for command and control. Experimental results on synthetic topologies embedded within Internet traffic traces from an ISP's backbone network indicate that our techniques (i) can localize the majority of bots with low false positive rate, and (ii) are resilient to incomplete visibility arising from partial deployment of monitoring systems and measurement inaccuracies from dynamics of background traffic.

## 1 Introduction

Malware is an extremely serious threat to modern networks. In recent years, a new form of general-purpose malware known as *bots* has arisen. Bots are unique in that they collectively maintain communication structures across nodes to resiliently distribute commands from a *command and control* (C&C) node. The ability to coordinate and upload new commands to bots gives the botnet owner vast power when performing criminal activities, including the ability to orchestrate surveillance attacks, perform DDoS extortion, sending spam for pay, and phishing. This problem has worsened to a point where modern botnets control hundreds of thousands of hosts and generate revenues of millions of dollars per year for their owners [23, 42].

Early botnets followed a centralized architecture. However, growing size of botnets, as well as the development of mechanisms that detect centralized command-and-control servers [10, 44, 27, 31, 72, 9, 49, 30, 29, 76], has motivated the design of decentralized peer-to-peer botnets. Several recently discovered botnets, such as Storm, Peacomm, and Conficker, have adopted the use of *structured* overlay networks [71, 57, 58]. These networks are a product of research into efficient communication structures and offer a number of benefits. Their lack of centralization means a botnet herder can join and control at any place, simplifying ability to evade discovery. The topologies themselves provide low delay any-to-any communication and low control overhead to maintain the structure. Further, structured overlay mechanisms are designed to remain robust in the face of churn [48, 32], an important concern for botnets, where individual machines may be frequently disinfected or simply turned off for the night. Finally, structured overlay networks also have protection mechanisms against active attacks [12].

In this work, we examine the question of whether ISPs can detect these efficient communication structures of peer-to-peer (P2P) botnets and use this as a basis for botnet defense. ISPs, enterprise networks, and IDSs have significant visibility into these communication patterns due to the potentially large number of paths between bots that traverse their routers. Yet the challenge is separating botnet traffic from background Internet traffic, as each botnet node combines command-and-control communication with the regular connections made by the machine's user. In addition, the massive scale of the communications makes it challenging to perform this task efficiently.

We propose BotGrep, an algorithm that isolates efficient peer-to-peer communication structures solely based on the information about which pairs of nodes communicate with one another (communication graph). Our approach relies on the fast-mixing nature of the structured P2P botnet C&C graph [26, 11, 6, 79]. The BotGrep algorithm iteratively partitions the communication graph into a faster-mixing and a slower-mixing piece, eventually narrowing on to the fast-mixing component. Although graph analysis has been applied to botnet and

P2P detection [15, 36, 78, 35], our approach exploits the spatial relationships in communication traffic to a significantly larger extent than these works. Based on experimental results, we find that under typical workloads and topologies our techniques localize 93-99% of botnet-infected hosts with a false positive probability of less than 0.6%, even when only a partial view of the communication graph is available. We also develop algorithms to run BotGrep in a privacy-preserving fashion, such that each ISP keeps its share of the communication graph private, and show that it can still be executed with access to a moderate amount of computing resources.

The BotGrep algorithm is content agnostic, thus it is not affected by the choice of ports, encryption, or other content-based stealth techniques used by bots. However, BotGrep must be paired with some sort of malware detection scheme, such as anomaly or misuse detection, to be able to distinguish botnet control structures from other applications using peer-to-peer communication. A promising approach starts with a honeynet that "traps" a number of bots. BotGrep is then able to take this small seed of bot nodes and recover the rest of the botnet communication structure and nodes.

*Roadmap:* We start by giving a more detailed problem description in Section 2. In Section 3, we describe our overall approach and core algorithms, and describe privacy-preserving extensions that enable sharing of observations across ISP boundaries in Section 4. We then evaluate performance of our algorithms on synthetic botnet topologies embedded in real Internet traffic traces in Section 5. We provide a brief discussion of remaining challenges in Section 6, and describe related work in Section 7. Finally, we conclude in Section 8.

## 2 System Architecture

In this section we describe several challenges involved in detecting botnets. We then describe our overall architecture and system design.

**Challenges:** Over the recent years, botnets have been adapting in order to evade detection and their activities have become increasingly stealthy. Botnets use random ports, encrypt their communication contents, thus defeating content-based identification. Traffic patterns, which have previously been used for detection [29], could potentially be altered as well, using content padding or other approaches. However, overall, it seems hard to hide the *fact* that two nodes are communicating, and thus we use this information as the basis for our design.

However, we are faced with several additional challenges. The background traffic on the Internet is highly variable and continuously changing, and likely dwarfs the small amount of control traffic exchanged between botnet hosts. Further, botnet nodes combine their malicious activity with the regular traffic of the legitimate users, thus they are deeply embedded inside the background communication topology. For example, Figure 1(b) shows a visualization of a synthetic P2P botnet graph embedded within a communication graph collected from the Abilene Internet2 ISP. The botnet is tightly integrated and cannot be separated from the rest of the nodes by a small cut.

In order to observe a significant fraction of botnet C&C traffic, it is necessary to combine observations from many vantage points across multiple ISPs. This creates an extremely large volume of data, since originally the background traffic will be captured as well. Thus, any analysis algorithms face a significant scaling challenge. In addition, although ISPs have already demonstrated their willingness to detect misbehavior in order to better serve their customers [3] as well as cooperating across administrative boundaries [4], they may be reluctant to share traffic observations, as those may reveal confidential information about their business operations or their customers.

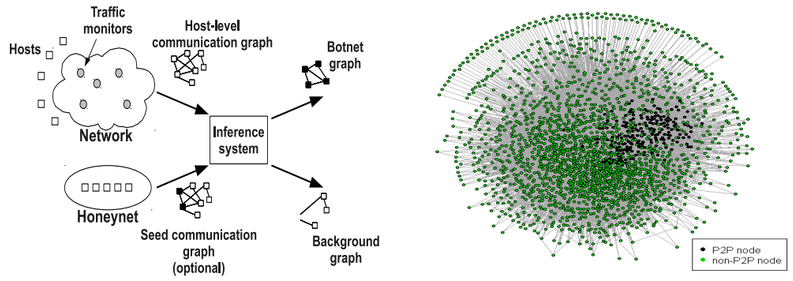
We next propose a botnet defense architecture that addresses these challenges.

**System architecture** : As a first step, our approach requires collecting a communication graph, where the nodes represent Internet hosts and edges represent communication (of any sort) between them. Portions of this graph are already being collected by various ISPs: the need to perform efficient accounting, traffic engineering and load balancing, detection of malicious and disallowed activity, and other factors, have already led network operators to deploy infrastructure to monitor traffic across multiple vantage points in their networks. BotGrep operates on a graph that is obtained by combining observations across these points into a single graph, which offers significant, though incomplete visibility into the overall communication of Internet hosts [1]. Traffic monitoring itself has been studied in previous work (e.g., [44]), and hence our focus in this work is not on architectural issues but rather on building scalable botnet detection algorithms to operate on such an infrastructure.

A second source of input is misuse detection. Since botnets use communication structures similar to other P2P networks, the communication graph alone may not

[1]Tools such as Cisco IOS's NetFlow [2] are designed to *sample* traffic by only processing one out of every 500 packets (by default). To evaluate the effect of sampling, we replayed packet-level traces collected by the authors of [42] from Storm botnet nodes, and simulated NetFlow to determine the fraction of botnet links that would be detected. We found that in the worst case (assuming each flow traversed a different router), after 50 minutes, 100% of botnet links were detected. Moreover, recent advances in counter architectures [77] may enable efficient tracking of the entire communication graph without need for sampling.






(a) (b)

Figure 1: (a) BotGrep architecture and (b) Abilene network with embedded P2P subgraph

be enough to distinguish the two. Some form of indication of malicious activity, such as botnet nodes trapped in Honeynets [68] or scanning behavior detected by Darknets [7], is therefore necessary. A list of misbehaving hosts can act as an initial "seed" to speed up botnet identification, or it can be used later to verify that the detected network is indeed malicious.

The next step is to isolate a botnet communication subgraph. Recently, botnet creators have been turning to communication graphs provided by structured networks, both due to their advantages in terms of efficiency and resilience, and due to easy availability of well-tested implementations of the structured P2P algorithms (e.g., Storm bases the C&C structure for its supernodes on the Overnet implementation of Kademlia [50]). One common feature of these structured graphs is their fast *mixing time*, i.e., the convergence time of random walks to a stationary distribution. Our algorithm exploits this property by performing random walks to identify fast-mixing component(s) and isolate them from the rest of the communication graph. If sharing of sensitive information is an issue, it is possible to perform random walks in a privacy-preserving fashion on a graph that is split among a collection of ISPs.

Once the botnet C&C structure is identified and confirmed as malicious, BotGrep outputs a set of *suspect hosts*. This list may be used to install blacklists into routers, to configure intrusion detection systems, firewalls, and traffic shapers; or as "hints" to human operators regarding which hosts should be investigated. The list may also be distributed to subscribers of the service, potentially providing a revenue stream. The overall architecture is shown in Figure 1(a).

## 3 Inference Algorithm

Our inference algorithm starts with a *communication graph* $G = (V, E)$ with $V$ representing the set of hosts observed in traffic traces and undirected edges $e \in E$ inserted between communicating hosts. Embedded within $G$ is a *P2P graph* $G_p \subset G$, and the remaining subgraph $G_n = G - G_p$ containing non-P2P communications. The goal of our algorithms is to reliably *partition* the input graph $G$ into $\{G_p, G_n\}$ in the presence of dynamic background traffic and with only partial visibility.

### 3.1 Approach overview

The main idea behind our approach is that, since most P2P topologies are much more highly structured than background Internet traffic, we can partition by detecting subgraphs that exhibit different topological patterns from each other or the rest of the graph. We do this by performing *random walks*, and comparing the relative mixing rates of the P2P subgraph structure and the rest of the communication graph. The subgraph corresponding to structured P2P traffic is expected to have a faster mixing rate than the subgraph corresponding to the rest of the network traffic. The challenge of the problem is to partition the graph into these two subgraphs when they are not separated by a small cut, and to do so efficiently for very large graphs.

Our approach consists of three key steps. Since the input graph could contain millions of nodes, we first apply a prefiltering step to extract a smaller set of candidate peer-to-peer nodes. This set of nodes contains most peer-to-peer nodes, as well as false positives. Next, we use a clustering technique based on the SybilInfer algorithm [21] to cluster only the peer-to-peer nodes, and remove false positives. The final step involves validating the result of our algorithms based on fast-mixing characteristics of peer-to-peer networks.

### 3.2 Prefiltering Step

The key idea in the prefiltering step is that for short random walks, the state probability mass associated with nodes in the fast-mixing subgraph is likely to be closer to the stationary distribution than nodes in the slow-mixing subgraph. Let $P$ be the transition matrix of the random walks. $P$ is defined as

$$P_{ij} = \begin{cases} \frac{1}{d_i} & \text{if } i \rightarrow j \text{ is an edge in G} \\ 0 & \text{otherwise} \end{cases}, \quad (1)$$

where $d_i$ denotes the degree of vertex $i$ in $G$.

The probability associated with each vertex after the short random walk of length $t$, denoted by $q^t$, can be be used as a metric to compare vertices and guide the extraction of the P2P subgraph. The initial probability distribution $q^0$ is set to $q^0_i = 1/|V|$, which means that the walk starts at all nodes with the equal probability. We can recursively compute $q^t$ as follows:

$$q^t = q^{t-1} \cdot P \quad (2)$$

Now, since nodes in the fast-mixing subgraph are likely to have $q^t$ values closer to the stationary distribution than nodes in the slow-mixing subgraph, and because the stationary distribution is proportional to node degrees, we can cluster nodes with homogeneous $\frac{q^t_i}{d_i}$ values. However, before doing so, we apply a transformation to dampen the negative effects of high-degree nodes on structured graph detection. High-degree nodes or hubs are responsible for speeding up the mixing rate of the non-structured subgraph $G_n$ and can reduce the relative mixing rate of $G_p$ as compared to $G_n$. The transformation filter is as follows:

$$s_i = \left(\frac{q^t_i}{d_i}\right)^{\frac{1}{r}}, \quad (3)$$

where $r$ is the dampening constant. We can now cluster vertices in the graph by using the *k-means* algorithm [47] on the set of values $s$. The *k-means* clustering algorithm divides the points in $s$ into $k$ ($k \ll |V|$) clusters such that the sum of squares $J$ from points to the assigned cluster centers is minimized.

$$J = \sum_{j=1}^{k} \sum_{i=1}^{|V|} \|s_i - c_j\|^2, \quad (4)$$

where $c_j$ is the center of cluster $j$. The within-cluster sum of squares for each cluster constitutes the cluster score. The parameter $k$ is chosen using the method of Pelleg and Moore [56]. Starting from a user specified minimum number of clusters $k = k_{min}$ we repeatedly compute k-means over our dataset by incrementing $k$ up to a maximum of $k_{max}$. We then select the best-scoring $k$ value. $k_{min}$ and $k_{max}$ correspond to the minimum and maximum number of possible botnets within the dataset. In our experiments, we used $k_{min} = 0$ and $k_{max} = 20$.

Each of the $k$ clusters corresponds to a set of nodes in $V_G$, so we may partition our graph into subgraphs $\{G_1, G_2, \ldots, G_k\}$. We must now confirm or reject the hypothesis that each of these subgraphs contains a structured P2P graph. Clustering helps speed up the superlinear components of the following algorithm; we may also be able to focus our attention on a particular subset of clusters if misuse detection is concentrated within them.

Note that we can use the sparse nature of the matrix $P$ to compute $q^t$ using Equation 2 very efficiently in $O(|E| \cdot t)$ time. The time and space complexity of Equation 3 is $O(|V|)$, while Equation 4 can be computed in $O(k \cdot |V|)$ iterations. Thus the prefiltering step is a very efficient mechanism to obtain a set of candidate P2P nodes, capable of operating on large node graphs.

### 3.3 Clustering P2P Nodes

The subgraphs computed by the above step are likely to contain P2P nodes, but they are also likely to contain some non-P2P nodes due to the "leakage" of random walks out of the structured subgraph. We perform a second pass over the each subgraph $G_l \in G_1, G_2, \ldots, G_k$ to remove weakly connected nodes.

We cluster P2P nodes by using the SybilInfer [21] framework. SybilInfer is a technique to detect Sybil identities in a social network graph; a key feature of SybilInfer is a sampling strategy to identify a good partition out of an extremely large space of possibilities ($2^V$). However, the detection algorithm used in SybilInfer relies on the existence of a small cut between the honest social network and the Sybil subgraph, and is thus not directly applicable to our setting. Next, we present a modified SybilInfer algorithm that is able to detect P2P nodes.

**1. Generation of Traces :** The first step of the clustering is the the generation of a set of random walks on the input graph. The walks are generated by performing a number $n$ of random walks, starting at each node in the graph. A special probability transition matrix is used, defined as follows:

$$P'_{ij} = \begin{cases} \min(\frac{1}{d_i}, \frac{1}{d_j}) & \text{if } i \rightarrow j \text{ is an edge in G} \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

This choice of transition probabilities ensures that the stationary distribution of the random walk is uniform over all vertices. The length of the random walk is $O(\log |V|)$, while the number of random walks per node (denoted by $n$), is a tunable parameter of the system. Only the start vertex and end vertex of each random walk are used by the algorithm, and this set of vertex pairs is called the *traces*, denoted by $T$.

**2. A probabilistic model for P2P nodes:** At the heart of our detection algorithm lies a model that assigns a probability to each subset of nodes of being P2P nodes. Consider any cut $X \subseteq V$ of nodes in the graph. We wish to compute the probability that the set of nodes X are all P2P nodes, given our set of traces T, i.e. $P(X = P2P|T)$. Through the application of Bayes theorem, we have an expression of this probability:

$$P(X = P2P|T) = \frac{P(T|X = P2P) \cdot P(X = P2P)}{Z = P(T)} \quad (6)$$

Note that we can treat $P(T)$ as a normalization constant $Z$, as it does not change with the choice of $X$. The prior probability $P(X = P2P)$ can be used to encode any further knowledge about P2P nodes (using honeynets), or can simply be set uniformly over all possible cuts. Our key theoretical task here is the computation of the probability $P(T|X = P2P)$, since given this probability, we can compute $P(X = P2P|T)$ using the Bayes theorem.

Our intuition in proposing a model for $P(T|X = P2P)$ is that for short random walks, the state probability mass for peer-to-peer nodes quickly approaches the stationary distribution. Recall that the stationary distribution of our special random walks is uniform, and thus, the state probability mass for peer-to-peer nodes should be homogeneous. We can classify the random walks in the trace T into two categories: random walks that end in the set $X$, and random walks that end in the set $\overline{X}$ (complementary set of nodes).

Using our intuition that for short random walks, the state probability mass associated with peer-to-peer nodes is homogeneous, we assign a uniform probability to all walks ending in the set $X$. On the other hand, we make no assumptions about random walks ending in the set $\overline{X}$ (in contrast to the original SybilInfer algorithm). Thus,

$$P(T|X = P2P) = \Pi_{w \in T} P(w|X = P2P), \quad (7)$$

where $w$ denotes a random walk in the trace. Now if the walk $w$ ends in vertex $a$ in $X$, then we have that

$$P(w|X = P2P) = \sum_{v \in X} \frac{N_v}{n \cdot |V|} \cdot \frac{1}{|X|}, \quad (8)$$

where $N_v$ denotes the number of random walks ending in vertex $v$. Observe that this probability is the same for all vertices in $X$. On the other hand, if the walk $w$ ends in vertex $a$ in $\overline{X}$, then we have that

$$P(w|X = P2P) = \frac{N_a}{n \cdot |V|}. \quad (9)$$

**3. Metropolis-Hastings Sampling:** Using the probabilistic model for P2P nodes, we have been able to compute the the probability $P(X = P2P|T)$ up to a multiplicative constant $Z$. However, computing $Z$ is difficult since it involves enumeration over all subsets X of the graph. Thus, instead of directly calculating this probability for any configuration of nodes $X$, we will sample configurations $X_i$ following this distribution. We use the Metropolis-Hastings algorithm [34] to compute a set of samples $X_i \sim P(X|T)$. Given a set of samples $S$, we can compute marginal probabilities of nodes being P2P nodes as follows:

$$P[i \text{ is } P2P] = \frac{\sum_{j \in S} I(i \in X_j)}{|S|}, \quad (10)$$

where $I(i \in X_j)$ is an indicator random variable taking value 1 if node $i$ is in the P2P sample $X_j$, and value 0 otherwise. Finally, we can use a threshold on the marginal probabilities (set to 0.5) to partition the set of nodes into fast-mixing and slow-mixing components.

### 3.4 Validation

We note that a general graph may be composed of multiple subgraphs having different mixing characteristics. However, our modified SybilInfer based clustering approach only partitions the graph into two subgraphs. This means we may have to use multiple iterations of the modified SybilInfer based clustering algorithm to get to the desired fastest mixing subgraph. This raises an important question - what is the termination condition for the iteration. In other words, we need a validation test to establish that we have obtained the fast-mixing P2P subgraph that we were trying to detect. Next, we propose a set of validation tests: if all of the tests are true, the iteration is terminated.

- Graph Conductance test: It has been shown [62] that the presence of a small cut in a graph results in a slow mixing time and that a fast-mixing time implies the absence of small cuts. To formalize the notion of a small cut, we use the measure of *graph conductance* ($\Phi_x$) [43] between cuts $(X, \overline{X})$, defined as

  $$\Phi_X = \frac{\Sigma_{x \in X} \Sigma_{y \notin X} \pi(x) P_{xy}}{\pi(X)}$$

  Since peer-to-peer networks are fast mixing, their graph conductance should be high (they do not have a small cut). Thus we can prevent further partitioning of a fast-mixing subgraph by testing that the graph conductance between the cuts is high.
- $q^{(t)}$ *entropy comparison test:* Random walks on structured homogeneous P2P graphs are characterized by high entropy state probability distributions.

This means that on a graph with $n$ nodes, a random walk of length $t \approxeq log|n|$ results in $q_i^{(t)} = 1/n$. In this sense they are theoretically optimal. We compute the relative entropy of the state probability distribution in graph $G(V,E)$ versus its theoretical optimal equivalent graph $G^T$. For this we use the Kullback-Leibler (KL) divergence measure [45] to calculate the relative entropy between $q_G$ and $q_{G^T}$: $F_G = \sum_x q_{G^T}(x) \log \frac{q_{G^T}(x)}{q_G(x)}$ When $F_G$ is close to zero then the mixing rates of $G$ and $G^T$ are comparable. This step can be computed in $O(|V|)$ time and $O(|V|)$ space.

- *Degree-homogeneity test:* The entropy comparison test above does not rule out fast-mixing heterogeneous graphs such as a star topology. However since structured P2P graphs have relatively homogeneous degree distributions (by definition), we need an additional test to measure the dispersion of degree values. In our study, we measured the coefficient of variation of the degree distribution of $G$, defined as the ratio of standard deviation and mean: $c_G = \sigma/\mu$. $c_G$ will be 0 for a fully homogeneous degree distribution. This metric can also be computed within $O(|V|)$ time and space.

# 4 Privacy Preserving Graph Algorithms

In general, ISPs treat the monitoring data they collect from their own networks as confidential, since it can reveal proprietary information about the network configuration, performance, and business relationships. Thus, they may be reluctant to share the pieces of the communication graph they collect with other ISPs, presenting a barrier to deploying our algorithms. In this section, we present privacy-preserving algorithms for performing the computations necessary for our botnet detection. Fundamentally, these algorithms support the task of performing a random walk across a distributed graph.

## 4.1 Establishing a Common Identifier Space

Our algorithms are expressed in terms of a graph $G = (V,E)$, where the vertices are Internet hosts and edges are connections between them. This graph is assembled from $m$ subgraphs belonging to $m$ ASes, $G_i = (V_i, E_i)$ such that $G = \bigcup_{i=1}^m G_i$. To simplify computations, we would like to generate an index mapping $I : \mathbb{Z}_{|V|} \to V$. We base our approach on private set intersection protocols. In particular, Jarecki and Liu have shown how to use Oblivious Pseudo-Random Functions (OPRFS) to perform private set intersection in linear time, i.e., $O(|V_i| + |V_j|)$. [37]. The basic approach consists of having a server pick a PRF $f_k(x)$, with a secret $k$. The server then evaluates $S = \{f_k(s_i)\}$ for all points within the server's set and sends it to the client. The client then, together with the server, evaluates the PRF obliviously on all $c_i$ for its own set; i.e, the client learns $C = \{f_k(c_i)\}$ without learning $k$, whereas the server learns nothing except $|C|$. The client can then compute $C \cup S$ and thus find the intersection.

We extend this approach to our problem as follows: we pick one AS to act as the server, and the rest as clients. Each client uses OPRF to compute $f_k(V_i)$. The server then generates an ordered list of $f_k(V_1)$ and sends it to the second AS. The second AS finds $f_k(V_1) \cap f_k(V_2)$ and thus identifies the positions of its nodes in the vector. It then appends $f_k(V_2)$ $f_k(V_1)$ to the list and sends the resulting list $f_k(V_1 \cup V_2)$ to the next AS. This process continues until the last AS is reached, who then reports $|V|$ to all of the others. Each AS can then compute $I$ for any node $v$ in its subgraph by finding the corresponding position of $f_k(v)$ in the list it saw.

Next, the ASes needs to eliminate duplicate edges. A similar algorithm can be used here, with each ISP dropping from its observations any edge that was also observed by another ISP that comes earlier in the list. Alternatively, routing information can be used to determine which edges might be observed by which other AS and perform a pairwise set intersection including only those nodes.

Finally, to perform random walk, each AS needs to learn the degree of each node. Since we eliminated duplicated edges, $d(v) = \sum_{i=1}^m d_i(v)$, where $d_i(v)$ is the degree of node $v$ in $G_i$. The sum can be computed by a standard privacy-preserving protocol, which is an extension of Chaum's dining cryptographer's protocol [13]. Each AS $i$ creates $m$ random shares $s_j^{(i)} \in \mathbb{Z}_l$ such that $\sum_{j=1}^m s_j^{(i)} \equiv d_i(v) \bmod l$ (where $l$ is chosen such that $l > \max_v d(v)$). Each share $s_j^{(i)}$ is sent to AS $j$. After all shares have been distributed, each AS computes $s_i = \sum_{j=1}^m s_i^{(j)} \bmod l$ and broadcasts it to all the other ASes. Then $d(v) = \sum_{i=1}^m s_i \bmod l$. This protocol is information-theoretically secure: any set of malicious ASes $S$ only learns the value $d(v) - \sum_{j \in S} d_i(v)$. The protocol can be executed in parallel for all nodes $v$ to learn all node degrees.

## 4.2 Random Walk

We perform a random walk by using matrix operations. In particular, given a transition matrix $T$ and an initial state vector $\vec{v}$, we can compute $T\vec{v}$, the state vector after a single random walk step. Our basic approach is to create matrices $T_i$ such that $\sum_{i=1}^m T_i = T$. We can then compute $T_i\vec{v}$ in a distributed fashion and compute the final sum at the end.

To construct $T_i$, an AS will set the value $(T_i)_{j,k}$ to be $1/d(v_j)$ for each edge $(j,k) \in E_i$ (after duplicate edges have been removed). Note that this transition matrix is sparse; it can be represented by $N$ linked lists of non-zero values $(T_i)_{j,k}$. Thus, the storage cost is $O(|E_i|) \ll O(|V_i|^2)$.

To protect privacy, we use Paillier encryption [55] to perform computation on an encrypted vector $E(\vec{v})$. Paillier encryption supports a homomorphism that allows one to compute $E(x) \oplus E(y) = E(x+y)$; it also allows the multiplication by a constant: $c \otimes E(x) = E(cx)$. This, given an encrypted vector $E(\vec{v})$ and a known matrix $T_i$, it is possible to compute $E(T_i\vec{v})$.

Damgård and Jurik [20] showed an efficient distributed key generation mechanism for Paillier that allows the creation of a public key $K$ such that no individual AS knows the private key, but together, they can decrypt the value. In the full protocol, one AS creates an encrypted vector $E(\vec{v})$ that represents the initial state of the random walk. This vector is sent to each AS, who then computes $E(T_i\vec{v})$. The ASes sum up the individual results to obtain $E(\sum_{i=1}^m T_i\vec{v}) = E(T\vec{v})$. This process can be iterated to obtain $E(T^k\vec{v})$. Finally, the ASes jointly decrypt the result to obtain $T^k\vec{v}$.

Note that Paillier operates over members $\mathbb{Z}_n$, where $n$ is the product of two large primes. However, the vector $\vec{v}$ and the transition matrices $T_i$ contain fractional values. To address this, we used fixed-point representation, storing $\lfloor x \times 2^c \rfloor$ (equivalently, $(x-\varepsilon) \times 2^c$, where $\varepsilon < 2^{-c}$). Each multiplication results in changing the position of the fixed point, since:

$$((x-\varepsilon_1) \times 2^c)((y-\varepsilon_2) \times 2^c) = (xy - \varepsilon_3) \times 2^{2c}$$

where $\varepsilon_3 < 2^{-c+1}$. Therefore, we must ensure that $2^{kc} < n$, where $k$ is the number of random walk steps. The maximal length random walk we use is $2\log_{\bar{d}}|V|$, where $\bar{d}$ is the average node degree, so $k < 40$, which gives us plenty of fixed-point precision to work with for a typical choice of $n$ (1024 or 2048 bits).[2]

## 4.3 Performance

Although the base privacy-preserving protocols we propose are efficient, due to the large data sizes, the operations still take a significant amount of processing time.

[2]Note that the multiplication of probabilities might result in values that are extremely small; however, the number of digits after the fixed point correspondingly increases after each multiplication, preventing loss of precision.

[3]The CPU time is estimated based on experiments on different hardware; however, these numbers are intended to provide an order-of-magnitude estimate of the costs.

Table 1: Privacy Preserving Operations

| Step | CPU time AS1 (s) [3] |
|---|---|
| 1. Determine common identifiers | 1 020 000 |
| 2. Eliminate duplicate edges | 8 160 000 |
| 3. Compute node degrees | *(no crypto)* |
| 4. Random walk (20 steps) | 8 000 000 |

We estimate the actual processing costs and bandwidth overhead, using some approximate parameters. In particular, we consider a topology of 30 million hosts, with an average degree of 20 per node.[4]

The running time of the intersections to compute a common representation is linear in $|V_i| + |V_j|$. We expect that $|V_i| < |V|$, but in the worst case, each ISP sees all of the nodes. Projecting linearly, we expect to spend about 30 000s on an intersection between two ISPs. Most ASes must perform only one intersection, but the first AS is involved in $m-1$ intersections. We expect $m$ to be around 35, based on our analysis of visibility of bot paths by tier-1 ISPs (Section 5.1). An important feature of the algorithm is that each ISP other than the first need only perform as many OPRF evaluations as it has nodes in its observation table, thus smaller ISPs with fewer resources need to perform correspondingly less work. We therefore suggest that the largest contributing ISP be chosen as the server. De Cristofaro and Tsudik suggest an efficiency improvement for Jaercki and Liu's algorithm [18]; they find that the server computation for 1 000 client values is less than $400ms$. Projecting linearly, we expect that the server load per client should be 12 000 seconds.

The next series of set intersections involve edge sets. The worst-case scenario for this computation assumes that all ASes see all edges, although, of course, this is unlikely (and would mean that the participation of some ASes is redundant). The load on the central server is $(0.4s/1000) \cdot 600\,000\,000 \cdot 34 = 8\,160\,000s$

A step of the random walk requires $O(|E|)$ homomorphic multiplications and additions of encrypted values. Our measurements with `libpaillier`[5] show that the multiplications are two orders of magnitude slower than additions. We were able to perform approx. 1500 multiplications per second using a 2048-bit modulus. This means that a single step would take 400 000s of computation.

We summarize the costs of the computation in Table 1. It is important to note that all of the operations are trivially parallelizable and thus can be computed on a moderately-sized cluster of commodity machines. Additionally, the table represents the costs of an initial computation; updated results can be computed by operating

[4]The choice of topology size and the average node degree is motivated from our experimental setting in Section 5.

[5]`http://acsc.cs.utexas.edu/libpaillier/`

only on the deltas of the observations, which we expect to be significantly smaller.

## 5 Results

To evaluate performance of our design, we evaluate it in the context of real Internet traffic traces. Ideally, to evaluate our design, we would like to have a list of all bots in the Internet, along with which logs of packets flowing between them, in addition to packet traces between non-botnet hosts. Unfortunately, acquiring data this extensive is very hard, due to the (understandable) reluctance of ISPs to share their internal traffic, and the difficulty in gaining ground truth on which hosts are part of a botnet.

To address this, we apply our approach to synthetic traces. In particular, we construct a topology containing a botnet communication graph, and embed it within a communication graph corresponding to background traffic. To improve realism, we build the background traffic communication graph by using real traffic collected from Netflow logs from the IP backbone of the Abilene Internet2 ISP. For our analysis, we consider a full day's trace collected on 22 October 2009. Since Abilene's NetFlow traces are aggregated into /24-sized subnets for anonymity, we perform the same aggregation for the botnet graph, and collect experimental results over the resulting subnet-level communication graph (we expect if our design were deployed in practice with access to per-host information, its performance would improve due to increased visibility). To investigate sensitivity of our results to this methodology and data set, we also use packet-level traces collected by CAIDA on OC192 Internet backbone links [5] on 11 January 2009. To construct the botnet graph, we select a random subset of nodes in the background communication graph to be botnet nodes, and synthetically add links between them corresponding to a particular structured overlay topology. We then pass the combined graph as input to our algorithm. By keeping track of which nodes are bots (this information is not passed to our algorithm), we can acquire "ground truth" to measure performance. To investigate sensitivity of our techniques to the particular overlay structure, we consider several alternative structured overlays, including (a) Chord, (b) de Bruijn, (c) Kademlia, and (d) the "robust ring" topology described in [39]. The remainder of this section contains results from running our algorithms over the joined botnet and Internet communication graphs, and measuring the ability to separate out the two from each other.

Before we proceed to the results, we first illustrate our inference algorithm with an example run.

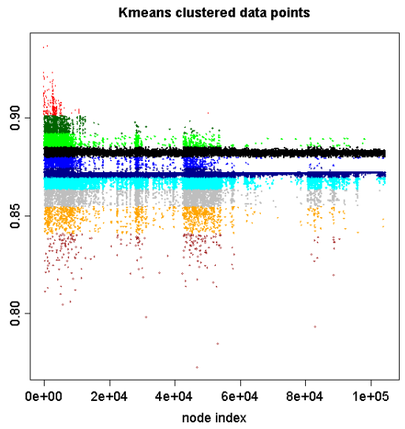



Figure 2: The filtered limit distribution ($s_i$) after clustering

### 5.1 Algorithm Example

Let us consider a specific application of our algorithm on a synthetically-generated de Bruijn [41] peer-to-peer graph embedded within a communication graph sampled from the Internet (using NetFlow traces from the Abilene Internet2 ISP). The Abilene communication graph $G_D$ contains $|V_D| = 104426$ nodes. We then generated a de Bruijn graph $G_p$ of 10000 nodes, with $m = 10$ outgoing links and $n = 4$ dimensions (10% of $|V|$). $G_p$ is then embedded in $G_D$ by mapping a node in $G_p$ into a node in $G_D$: for every node $i \in V_B$ we select a node $j \in V_D$ uniformly at random between 1 and $|V_D|$ without replacement, and add the corresponding edges in $E_B$ to $E_D$. The resulting graph is $G(V,E)$ with $N = |V| = 104426$ nodes and $|E| = 647053$ edges. The goal of our detection technique is to extract $G_p$ from $G_D$ as accurately as possible.

First, we apply the pre-filtering step: we carry out a short random walk starting from every node with probability $1/N$ to obtain $q^{(t)}$, on which the transformation filter of Equation 3 is applied to obtain $s$. We used a dampening constant of $r = 100$ to undermine the influence of hub nodes on the random walk process. The data points in $s$ corresponding to each of the partitions returned by k-means clustering is shown in Figure 2.

In the example we consider here, applying the k-means algorithm gives us ten sets of potential P2P candidates. In a completely unsupervised setting, we would need to run the modified SybilInfer algorithm on each of the candidate sets. However we expect that the analysis can simply be focused on the candidate set containing the set of honey-net nodes. Thus, let us consider the graph



Table 2: Termination Conditions

| Condition | Final iter. | Other iters. |
|---|---|---|
| Conductance | 0.9 | $< 0.5$ |
| KL-divergence | 0.1 | $> 0.45$ |
| Entropy | 0.97 | $< 0.64$ |
| Coeff. of variation | $< 1$ | $> 4.6$ |

nodes corresponding to the fourth cluster (colored in yellow). The cluster size is 17576 nodes.

Next, we recursively apply the modified SybilInfer partitioning algorithm to this cluster. After three iterations of the SybilInfer partitioning algorithm, we obtain a subgraph of size 10143 nodes, containing 9905 P2P nodes, and 238 other nodes. At this stage, our set of validation conditions indicates that the sub-graph is indeed fast mixing, and we stop the recursion. Table 2 shows the values of the validation metrics on the final subgraph and the previous graphs. There is a significant gap, making it easy to select a threshold value.

To evaluate performance, we are concerned with the *false positive rate* (the fraction of non-bot nodes that are detected as bots) and the *false negative rate* (the fraction of bot nodes that are not detected). These results are shown in Tables 3(a) and 3(b). The experimental methodology and parameters used were the same as in the above example. All results are averaged over five random seeds. Overall, we found that BotGrep was able to detect 93-99% of bots over a variety of topologies and workloads. In particular, we observed several key results:

**Effect of botnet topology:** To study applicability of our approach to different botnet topologies, we consider Kademlia [50], Chord [70], and de Bruijn graphs. In addition, we also consider the LEET-Chord topology [39], a recently proposed overlay topology that aims to be difficult to detect (cannot be reliably detected with existing traffic dispersion graph techniques). Overall, we find performance to be fairly stable across multiple kinds of botnet topologies, with detection rates higher than 95%. In addition, BotGrep is able to achieve a false positive rate of less than 0.42% on the harder-to-detect LEET-Chord topology. While our approach is not perfectly accurate, we envision it may be of use when coupled with other detection strategies (e.g., previous work on botnet detection [38, 36], or if used to signal "hints" to network operators regarding which hosts may be infected. Furthermore, while the LEET-Chord topology is harder to detect, this comes at a tradeoff with less resilience to failure. To study the robustness of the LEET-Chord topology, Figure 3 shows the robustness of Chord and LEET-Chord by randomly removing varying percentages of nodes. We observed that LEET-Chord is much less resilient to node failures (or active attacks) as compared with Chord. This trade-off between stealthiness of the

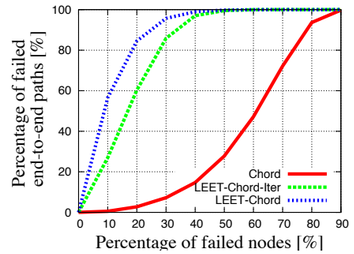



Figure 3: Robustness of Chord and LEET-Chord with 65,536 nodes. We also consider an alternative LEET-Chord-Iter, where routing proceeds as in regular LEET-Chord, but when the destination is outside the node's cluster, and when all long range links are failed, it greedily forwards the packet iteratively to next clockwise cluster.

topology and its resilience is not surprising, since a common indicator of resilience is the bisection bandwidth, and Sinclair [66] has shown that the bisection bandwidth is bounded by the mixing time of the topology. Thus, it is likely that the use of stealthy slow mixing topologies to escape detection via BotGrep would adversely effect the resilience of the botnet.

**Effect of botnet graph size:** Next, we vary the size of the embedded botnet. We do this to investigate performance as a function of botnet size, for example, to evaluate whether BotGrep can efficiently detect small botnets (e.g., bots in early stages of deployment, which may have greater chance of containment) and large-scale botnets (which may pose significant threats due to their size and large topological coverage). We perform this experiment by keeping the size of the background traffic graph constant, and generating synthetic botnet topologies of varying sizes (between 100 and 100,000 bots). The degree of bot nodes in the case of Chord and Kademlia depend on the size of the topology ($\log N$), while for de Bruijn, we used a constant node degree of 10. Overall, we found that as the size of the bot graph increases, performance degrades, but only by a small amount. For example, in Table 3(a), with the fully visible de Bruijn topology, for 100 nodes the false positive rate is zero, while for 10,000 nodes the rate becomes 0.12%.

**Effect of background graph size:** One concern is that BotGrep may perform less accurately with larger background graphs, as it may become easier for the botnet structure to "hide" in the increasing number of links in the graph. To evaluate sensitivity of performance to scale, we vary the size of the background communication graph, by evaluating over both the Abilene and CAIDA dataset (104,426 and 3,839,936 nodes, respectively). To

(a) Abilene

| Topology | $\|V_B\|$ | % FP | % FN | % Detected |
|---|---|---|---|---|
| de Bruijn | 100 | 0.00 | 2.00 | 98.00 |
| | 1000 | 0.01 | 2.40 | 97.60 |
| | 10000 | 0.12 | 2.35 | 97.65 |
| Kademlia | 100 | 0.00 | 3.20 | 97.80 |
| | 1000 | 0.01 | 2.48 | 98.52 |
| | 10000 | 0.10 | 2.12 | 97.88 |
| Chord | 100 | 0.00 | 3.00 | 97.00 |
| | 1000 | 0.01 | 2.32 | 97.68 |
| | 10000 | 0.08 | 1.94 | 98.06 |
| LEET-Chord | 100 | 0.00 | 3.00 | 97.00 |
| | 1000 | 0.03 | 1.60 | 98.40 |
| | 10000 | 0.42 | 1.00 | 99.00 |

(b) CAIDA

| Topology | $\|V_B\|$ | % FP | % FN | % Detected |
|---|---|---|---|---|
| de Bruijn | 1000 | 0.00 | 1.80 | 98.20 |
| | 10000 | 0.01 | 0.93 | 99.07 |
| | 100000 | 0.09 | 0.67 | 99.33 |
| Kademlia | 1000 | 0.00 | 2.10 | 97.90 |
| | 10000 | 0.01 | 0.80 | 99.20 |
| | 100000 | 0.19 | 0.17 | 99.83 |
| Chord | 1000 | 0.00 | 2.20 | 97.80 |
| | 10000 | 0.01 | 0.48 | 99.52 |
| | 100000 | 0.06 | 0.46 | 99.54 |
| LEET-Chord | 1000 | 0.00 | 0.40 | 99.60 |
| | 10000 | 0.02 | 0.48 | 99.52 |

Table 3: Detection and error rates of inference for (a) Abilene and (b) CAIDA communication graphs

(a) CAIDA 30M

| Topology | $\|V_B\|$ | % FP | % FN | % Detected |
|---|---|---|---|---|
| de Bruijn | 100000 | 0.01 | 0.8 | 99.20 |
| Kademlia | 100000 | 0.01 | 0.4 | 99.60 |
| Chord | 100000 | 0.01 | 0.4 | 99.60 |

(b) Leveraging Honeynets - CAIDA

| Topology | $\|V_B\|$ | % FP | % FN | % Detected |
|---|---|---|---|---|
| de Bruijn | 100000 | 0.04 | 0.8 | 99.20 |
| Kademlia | 100000 | 0.05 | 0.4 | 99.60 |
| Chord | 100000 | 0.04 | 0.4 | 99.60 |

Table 4: Detection and error rates of inference (a) for CAIDA 30M (b) when leveraging Honeynets for CAIDA.

get a rough sense of performance on much larger background graphs, we also build a "scaled up" version of the CAIDA graph containing 30 million hosts while retaining the statistical properties of the CAIDA graph. To scale up the CAIDA graph $G_c$ by a factor of $k$, we make $k$ copies of $G_c$, namely $G_1 \ldots G_k$ with vertex sets $V_1 \ldots V_k$ and edge sets $E_1 \ldots E_k$. Note that for each edge $(p,q)$ in $E_r$, we have a corresponding edge in each copy $G_1 \ldots G_k$, we refer to these as $(p_1,q_1) \ldots (p_k,q_k)$. We then compute the graph disjoint union over them as $G_S(V_S,E_S)$ where $V_S = (V_1 \cup V_2 \cdots \cup V_k$ and $E_S = E_1 \cup E_2 \cdots \cup E_k)$. Next, we randomly select a fraction of links from $E_S$ to obtain a set of edges $E_r$ that we shall rewire. As a heuristic, we set the number of links selected for rewiring to $|E_r| = k\sqrt{N\log(N)}$ where $N$ is the number of nodes in the CAIDA graph $G_c$. For each edge $(p,q)$ in $E_r$ we wish to rewire, we choose two random numbers $a$ and $b$ ($1 \le a,b \le k$) and rewire edges $(p_a,q_a)$ and $(p_b,q_b)$ to $(p_a,q_b)$ and $(p_b,q_a)$ such that $d_{p_a} = d_{p_b}$ and $d_{q_a} = d_{q_b}$. This edge rewiring ensures that (a) the degree of all four nodes $p_a$,$q_a$,$p_b$ and $q_b$ remains unchanged, (b) the joint degree distribution $P(d_1,d_2)$ – the probability that an edge connects $d_1$ and $d_2$ degree nodes remains unchanged, and (c) $P(d_1,d_2,\ldots d_l)$ remains unchanged as well, where $l$ is the number of unique degree values that nodes in $G_c$ can take.

Overall, we found that BotGrep scales well with network size, with performance remaining stable as network size increases. For example, in the CAIDA dataset with a background graph of size 3.8 million hosts, the false positive rate for the de Bruijn topology of size 100000 is 0.09% (shown in Table 3b), while for the scaled up 30 million node CAIDA topology, this rate is 0.01 (Table 4(a)). Observe that the false positive rate has decreased by a factor of 9, which is approximately equal to the scale up factor between the two topologies, indicating the the actual number of false positives remains the same. This indicates that the number of false positives depend on botnet size and not the background graph size.

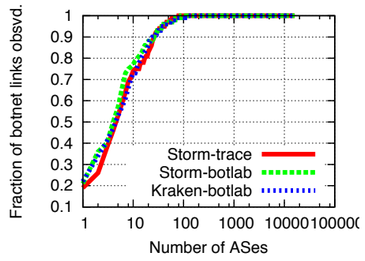



Figure 4: Number of visible botnet links, as a function of number of most-affected ASes contributing views.

**Effect of reduced visibility:** In the experiments we have performed so far, the embedded structured graph $G_p$ is present in its entirety. However, just as $G_D$ is obtained by sampling Internet or enterprise traffic, only a subset of botnet control traffic will actually be available to us. It is therefore important to evaluate how well our algorithms work with graphs where only a fraction of the structured subgraph edges are known. To study this, we evaluate performance of our scheme when deployed at only a subset of ISPs in the Internet. To do this, we collected



(a) Abilene

| Topology | $\|V_B\|$ | % FP | % FN | % Detected |
|---|---|---|---|---|
| de Bruijn | 100 | 0.00 | 3.00 | 97.00 |
| | 1000 | 0.02 | 2.80 | 97.20 |
| | 10000 | 0.17 | 3.31 | 96.69 |
| Kademlia | 100 | 0.00 | 3.75 | 96.25 |
| | 1000 | 0.01 | 2.90 | 97.10 |
| | 10000 | 0.19 | 2.07 | 97.93 |
| Chord | 100 | 0.00 | 9.00 | 91.00 |
| | 1000 | 0.02 | 3.50 | 96.50 |
| | 10000 | 0.13 | 2.54 | 97.46 |
| LEET-Chord | 100 | 0.00 | 6.00 | 94.00 |
| | 1000 | 0.06 | 2.70 | 97.30 |
| | 10000 | 0.58 | 1.80 | 98.20 |

(b) CAIDA

| Topology | $\|V_B\|$ | % FP | % FN | % Detected |
|---|---|---|---|---|
| de Bruijn | 1000 | 0.00 | 2.70 | 97.30 |
| | 10000 | 0.00 | 4.22 | 95.78 |
| | 100000 | 0.12 | 1.74 | 98.26 |
| Kademlia | 1000 | 0.00 | 0.50 | 99.50 |
| | 10000 | 0.01 | 0.30 | 99.70 |
| | 100000 | 0.09 | 0.53 | 99.47 |
| Chord | 1000 | 0.00 | 3.40 | 96.60 |
| | 10000 | 0.01 | 0.65 | 99.35 |
| | 100000 | 0.06 | 5.36 | 94.64 |
| LEET-Chord | 1000 | 0.01 | 0.20 | 99.80 |
| | 10000 | 0.02 | 1.09 | 98.91 |

Table 5: Results if only Tier-1 ISPs contribute views, for (a) Abilene and (b) CAIDA

roughly 4,000 Storm botnet IP addresses from Botlab [1] (*botlab-storm*), and measured what fraction of inter-bot paths were visible from tier-1 ISPs. From an analysis of the Internet AS-level topology [63], we find that 60% of inter-bot paths traverse tier-1 ISPs. We found that if the most-affected ASes cooperate—the ASes with the largest number of bots—this number increased to 89%). Figure 4 shows this result in more detail. Here, we vary the number of ASes cooperating to contribute views (assuming the most-affected ASes contribute views first), plotting the number of visible inter-bot links. We repeat the experiment also for the Kraken botnet trace from [1] (*kraken-botlab*), as well as a packet-level trace from the Storm botnet (*storm-trace*). We find that if only the 5 most-affected ASes contribute views, 57% of Storm links and 65% of Kraken links were visible.

We therefore removed 40% of links from our botnet graphs (Table 5a and Table 5b). While the false-negative rate increases, our approach still detects over 90% of botnet hosts with high reliability (the false positive rate for the hard to detect LEET-Chord topology still remains less than 0.58%). Disabling or removing such a large fraction of nodes will lead to certain loss of operational capability.

**Leveraging Honeynets:** We shall now present an extension to our inference algorithm that leverages the knowledge of a few known bot nodes. This extension considers random walks starting only from the honeynet nodes to obtain a set of candidate P2P nodes in the prefiltering stage. Using this extension, we find that there is a significant gain in terms of reducing the false positives, as well as speeding up the efficiency of the protocol. As Table 4b shows, the false positive rate for the Kademlia topology has been reduced by a factor of 4 as compared to corresponding value in Table 3b. Furthermore, only a single iteration of the modified SybilInfer algorithm was required to obtain the final subgraphs, providing a significant gain in efficiency.

**Effect of inference algorithm:** For comparison purposes, we also consider several graph partitioning algorithms that have been proposed in the literature. While these techniques were not intended to scale up to the large data sets we consider here, we can compare against them on smaller data sets to get a sense of how BotGrep compares against these approaches. In particular, several algorithms for *community detection* (detecting groups of nodes in a network with dense internal connections) have been proposed. Work in this space mainly focuses on *hierarchical clustering methods*. Work in this space can be classified as following two categories, and for our evaluation we implement two representative algorithms from each category:

*Edge importance based* community structure detection iteratively removes the edges with the highest *importance*, which can be defined in different ways. Girvan and Newman [25] defined edge importance by its shortest path betweenness. The idea is that the edge with higher betweenness is typically responsible for connecting nodes from different communities. In [22], *information centrality* has been proposed to measure the edge importance. The information centrality of an edge is defined as the relative *network efficiency* [46] drop caused by the removal of that. The time complexity of algorithm in [25] and [22] are $O(|V|^3)$ and $O(|E|^3 \times V)$, respectively.

*The spectral-based* approach detects communities by optimizing the modularity (a benefit function measures community structure [52] over possible network divisions. In [53], the communities are detected by calculating the eigenvector of the modularity matrix. It takes $O(|E|+|V|^2)$ time to separating each community. Moreover, Clauset *et al.* [14] proposed a hierarchical agglomeration algorithm for community detecting. The proposed greedy algorithm adopts more sophisticated data structures to reduce the computation time of modularity calculation. The time complexity is $O(|E|+|V|\log_2|V|)$ in average.

As the time complexity of above algorithms is not acceptable for computing large-scale networks, here we

| Topology | BotGrep | Fast Greedy Modularity | Girvan-Newman Betweenness | Modularity Eigenvector |
|---|---|---|---|---|
| de Bruijn | 0.78/2.55 | 14.43/7.65 | 19.73/15.31 | 0.92/43.88 |
| Chord | 0.77/7.15 | 7.58/10.13 | 6.05/19.50 | 4.24/20.19 |
| Kademlia | 0.92/7.00 | 14.66/33.80 | 18.06/4.75 | 5.70/48.70 |

Table 6: 2k Abilene Results (% FP /% FN)

consider a small-scale scenario for performance evaluation. We extract subgraphs from full Abilene data by performing a Breadth-First-Search (BFS) starting at a randomly selected node, in which the overall visited nodes are limited by a size of 2000. Results from our comparison are shown in Table 6. The information centrality algorithm took more than one month to run for just one iteration on this 2000-node graph, and was hence excluded from further analysis (we tested information centrality on smaller 50-node graphs, and found performance comparable to the Girvan and Newman Betweenness algorithm). Overall, we found that our approach outperformed these approaches. For example, on the Chord topology, BotGrep's false positive rate was 0.77%, while false positive rates for the other approaches ranged from 4.24-7.58%. The performance of BotGrep is less on this scaled down 2000-node topology as compared to the earlier Abilene and CAIDA datasets, because our method of generating the scaled-down 2000 node graph selected the densely connected core of the graph, which is fast-mixing, while on more realistic graphs, it is easier for BotGrep to distinguish the fast-mixing botnet topology from the rest of the non-fast-mixing background graph.

Moreover, we found that run-time was a significant limiting factor in using these alternate approaches. For example, the Girvan-Newman Betweenness Algorithm took 2.5 hours to run on a graph containing 2000 nodes (in all cases, BotGrep runs in under 10.4 seconds on a Core2 Duo 2.83GHz machine with 4GB RAM using a single core). While these traditional techniques were not intended to scale to the large data sets we consider here, they may be appropriate for localizing smaller botnets in contained environments (e.g., within a single Honeynet, or the part of a botnet contained within an enterprise network). Since these techniques leverage different features of the inputs, they are synergistic with our approach, and may be used in conjunction with our technique to improve performance.

## 6 Discussion

As we have demonstrated, analysis of core Internet traffic can be effective at identifying nodes and communication links of structured overlay networks. However, many challenges remain to turn our approach into a full-scale detection mechanism.

**Misuse Detection:** It is easy to see that other forms of P2P activity, such as file sharing networks, will also be identified by our techniques. While there is some benefit to being able to identify such traffic as well, it requires a dramatically different response than botnets and so it is important to distinguish the two. We believe that fundamentally, our mechanisms need to be integrated with detection mechanisms at the edge that identify suspicious behavior. Also, multiple intrusion detection approaches can reinforce each other and provide more accurate results [75, 67, 30]; e.g., misbehaving hosts that follow a similar misuse pattern and at the same time are detected to be part of the same botnet communication graph may be precisely labeled as a botnet, even if each individual misbehavior detection is not sufficient to provide a high-confidence categorization.

A concrete example of how misuse detection may work is the following: we randomly sample nodes from the suspect P2P network and compute the likelihood of the sampled nodes being malicious, based on inputs from honeynets, spam blacklists etc. If we can identify a statistically significant difference of the rates of misuse, then we can assume that membership in the P2P network is correlated with misuse and we should label it as a P2P botnet. Note that, given the availability of large sample sizes, even a small difference in the rates will be statistically significant, so this approach will be successful even if misuse detection fails to identify the vast majority of the botnet nodes as malicious.

**Scale and cooperation:** Our experiments show our design can scale to large traffic volumes, and in the presence of partial observations. However, several practical issues remain. First, large ISPs tend to use sampled data analysis to monitor their networks. This can miss low-volume control communications used by botnet networks. New counter architectures or programmable monitoring techniques should be used to collect sufficient statistics to run our algorithms [73]. Also, for best results multiple vantage points should contribute data to obtain a better overall perspective.

**Tradeoffs between structure and detection:** The communication structure of botnet graphs plays an important role in their delay penalty, and how resilient they are to network failures. At the same time, our results indicate that the structure of the communication graph has some effect on the ability to detect the botnet host from a collection of vantage points. As part of future work, we plan to study the tradeoff between resilience and the ability to avoid detection, and whether there exist fundamentally hard-to-detect botnet structures that are also resilient.

**Containing botnets:** The ability to quickly localize structured network topologies may assist existing systems that monitor network traffic to quickly localize and contain bot-infected hosts. When botnets are detected in edge networks, the relevant machines are taken offline. However, this may not always be easy with in-core detection; an interesting question is whether in-core filtering or distributed blacklisting can be an effective response strategy when edge cooperation is not possible. Another question we plan to address is whether there exist responses that do not completely disconnect a node but mitigate its potential malicious activities, to be effected when a node is identified as a botnet member, but with a low confidence.

## 7 Related Work

The increasing criticality of the botnet threat has led to vast amounts of work that attempt to localize them. We can classify this work into host based approaches and network based approaches. Host based approaches detect intrusions by analyzing information available on a single host. On the other hand, network based approaches detect botnets by analyzing incoming and outgoing host traffic. Hybrid approaches exist as well. BotGrep (our work) is a network based approach to botnet detection that uses graph theory to detect botnets.

In the following section (Section 7.1) we review related work on network based approaches and then describe work on botnet detection using graph analysis (Section 7.2).

### 7.1 Network based approaches

Several pieces of work isolate bot-infected hosts by detecting the malicious traffic they send, which may be divided into schemes that analyze *attack traffic*, and schemes that analyze *control traffic*.

**Attack traffic:** For example, network operators may look for sources of denial of service attacks, port scanning, spam, and other unwanted traffic as a likely bot. These works focus on the symptoms caused by the botnets instead of the networks themselves. Several works seek to exploit DNS usage patterns. Dagon et al. [19] studied the propagation rates of malware released at different times by redirecting DNS traffic for bot domain names. Their use of DNS sinkholes is useful in measuring new deployments of a known botnet. However, this approach requires a priori knowledge of botnet domain names and negotiations with DNS operators and hence does not target scaling to networks where a botnet can simply change domain names, have a large pool of C&C IP addresses and change the domain name generation algorithm by remotely patching the bot. Subsequently, Ramachandran et al. [61] use a graph based approach to isolate spam botnets by analyzing the pattern of requests to DNS blacklists maintained by ISPs. They observed that legitimate email servers request blacklist lookups and are looked up by other email servers according to the timing pattern of email arrival, while bot-infected machines are a lot less likely to be looked up by legitimate email servers. However, DNS blacklists and phishing blacklists [65], while initially effective have are becoming increasingly ineffective [60] owing to the agility of the attackers. Much more recently, Villamar et al. [74] applied Bayesian methods to isolate centralized botnets that use fast-flux to counter DNS blacklists, based on the similarity of their DNS traffic with a given corpus of known DNS botnet traces. Further, in order to study bots, Honeypot techniques have been widely used by researchers. Cooke et al. [17] conducted several studies of botnet propagation and dynamics using Honeypots; Barford and Yegneswaran [8] collected bot samples and carried out a detailed study on the source code of several families; finally, Freiling et al. [24] and Rajab et al. [59] carried out measurement studies using Honeypots. Collins et al. [16] present a novel botnet detection approach based on the tendency of unclean networks to contain compromised hosts for extended periods of time and hence acting as a *natural* Honeypot for various botnets. However Honeypot-based approaches are limited by their ability to attract botnets that depend on human action for an infection to take place, an increasingly popular aspect of the attack vector [51].

**Control traffic:** Another direction of work, is to localize botnets solely based on the control traffic they use to maintain their infrastructures. This line of work can be classified as *traffic-signature* based detection and *statistical traffic analysis* based detection. Techniques in the former category require traffic signatures to be developed for every botnet instance. This approach has been widely used in the detection of IRC-based botnets. Blinkley and Singh[10] combine IRC statistics and TCP work weight to generate signatures; Karasaridis et al. [44] present an algorithm to detect IRC C&C traffic signatures using Netflow records; Rishi [27] uses n-gram analysis to identify botnet nickname patterns. The limitations of these approaches are analogous to the scalability issues faced by host-based detection techniques. In addition, such signatures may not exist for P2P botnets. In the latter category, several works [31, 72, 9, 49] suggest that bot-

nets can be detected by analyzing their flow characteristics. In all these approaches, the authors use a variety of heuristics to characterize the network behavior of various applications and then apply clustering algorithms to isolate botnet traffic. These schemes assume that the statistical properties of bot traffic will be different from *normal* traffic because of synchronized or correlated behavior between bots. While this behavior is currently somewhat characteristic of botnets, it can be easily modified by botnet authors. As such it does not derive from the fundamental property of botnets.

Other works use a hybrid approach such as Bothunter [30] which automates traffic-signature generation by searching for a series of flows that match the infection life-cycle of a bot; BotMiner [29] combines packet statistics of C&C traffic with those of attack traffic and then applies clustering techniques to heuristically isolate botnet flows. TAMD [76] is another method that exploits the spatial and temporal characteristics of botnet traffic that emerges from multiple systems within a vantage point. They aggregate flows based on similarity of flow sizes and host configuration (such as OS platforms) and compare them with a historical baseline to detect infected hosts.

Finally, there are also schemes that combine network- and host-based approaches. The work of Stinson et al. [69] attempts to discriminate between locally-initiated versus remotely-initiated actions by tracking data arriving over the network being used as system call arguments using taint tracking methods. Following a similar approach, Gummadi et al. [33] whitelist application traffic by identifying and attesting human-generated traffic from a host which allows an application server to selectively respond to service requests. Finally, John et al. [40] present a technique to defend against spam botnets by automating the generation of spam feeds by directing an incoming spam feed into a Honeynet, then downloading bots spreading through those messages and then using the outbound spam generated to create a better feed. While all the above are interesting approaches they again deal with the side-effects of botnets instead of tackling the problem in its entirety in a scalable manner.

### 7.2 Graph-based approaches

Several works [15, 36, 35, 78, 38] have previously applied graph analysis to detect botnets. The technique of Collins and Reiter [15] detects anomalies induced in a graph of protocol specific flows by a botnet control traffic. They suggest that a botnet can be detected based on the observation that an attacker will increase the number of connected graph components due to a sudden growth of edges between unlikely neighboring nodes. While it depends on being able to accurately model valid network growth, this is a powerful approach because it avoids depending on protocol semantics or packet statistics. However this work only makes minimal use of spatial relationship information. Additionally, the need for historical record keeping makes it challenging in scenarios where the victim network is already infected when it seeks help and hasn't stored past traffic data, while our scheme can be used to detect pre-existing botnets as well. Illiofotou et al. [36, 35] also exploit dynamicity of traffic graphs to classify network flows in order to detect P2P networks. It uses static (spatial) and dynamic (temporal) metrics centered on node and edge level metrics in addition to the largest-connected-component-size as a graph level metric. Our scheme however starts from first principles (searching for expanders) and uses the full extent of spatial relationships to discover P2P graphs including the joint degree distribution and the joint-joint degree distribution and so on.

Of the many botnet detection and mitigation techniques mentioned above, most are rather ad-hoc and only apply to specific scenarios of centralized botnets such as IRC/HTTP/FTP botnets, although studies [28] indicate that the centralized model is giving way to the P2P model. Of the techniques that do address P2P botnets, detection is again dependent on specifics regarding control traffic ports, network behavior of certain types of botnets, reverse engineering botnet protocols and so on, which limits the applicability of these techniques. Generic schemes such as BotMiner [29] and TAMD [76] using behavior based clustering are better off but need access to extensive flow information which can have legal and privacy implications. It is also important to think about possible defenses that botmasters can apply, the cost of these defenses and and how they might affect the efficiency of detection. Shear and Nicol [64, 54] describe schemes to mask the statistical characteristics of real traffic by embedding it in synthetic, encrypted, cover traffic. The adoption of such schemes will only require minimal alterations to existing botnet architectures but can effectively defend against detection schemes that depend on packet level statistics including BotMiner and TAMD.

## 8 Conclusion

The ability to localize structured communication graphs within network traffic could be a significant step forward in identifying bots or traffic that violates network policy. As a first step in this direction, we proposed BotGrep, an inference algorithm that identifies botnet hosts and links within network traffic traces. BotGrep works by searching for structured topologies, and separating them from the background communication graph. We give an architecture for a BotGrep network deployment as well as a privacy-preserving extension to simplify deployment across networks. While our techniques do not achieve perfect accuracy, they achieve a low enough false positive rate to be of substantial use, especially when combined with complementary techniques. There are several avenues of future work. First, performance of our approach may be improved by leveraging temporal information (observing how parts of the the communication graph change over time) to assist in separating out the botnet graph. In addition, it may be desirable to distinguish other peer-to-peer structure from other Internet background traffic, perhaps by observing more fine-grained properties of communication patterns. Finally, we do not attempt to address the challenging problem of botnet *response*. Future work may leverage our inferred botnet topologies by dropping crucial links to partition the botnet, based on the structure of the botnet graph.



## Acknowledgments

We would like to thank Vern Paxson and Christian Kreibich for sharing their Storm traces. We are also grateful to Reiner Sailer and Mihai Christodorescu for helpful discussions. This work is supported in part by National Science Foundation Grants CNS 06–27671 and CNS 08–31653.

## References

[1] Botlab: A real-time botnet monitoring platform. botlab.cs.washington.edu.

[2] Cisco IOS Netflow. http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[3] Comcast constant guard. http://security.comcast.net/constantguard/.

[4] Spamhaus. www.spamhaus.org.

[5] The Cooperative Association for Internet Data Analysis (CAIDA). http://www.caida.org/.

[6] J. Aspnes and U. Wieder. The expansion and mixing time of skip graphs with applications. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 126–134, New York, NY, USA, 2005. ACM Press.

[7] M. Bailey, E. Cooke, F. Jahanian, N. Provos, K. Rosaen, and D. Watson. Data Reduction for the Scalable Automated Analysis of Distributed Darknet Traffic. In *Proceedings of IMC*, 2005.

[8] P. Barford and V. Yegneswaran. *An Inside Look at Botnets*, volume 27 of *Advanced in Information Security*, chapter 8, pages 171–192. Springer, 2006.

[9] A. Barsamian. Network characterization for botnet detection using statistical-behavioral methods. Masters thesis, Thayer School of Engineering, Dartmouth College, USA, June 2009.

[10] J. R. Binkley and S. Singh. An algorithm for anomaly-based botnet detection. In *SRUTI'06: Proceedings of the 2nd conference on Steps to Reducing Unwanted Traffic on the Internet*, pages 7–7, Berkeley, CA, USA, 2006. USENIX Association.

[11] N. Borisov. *Anonymous routing in structured peer-to-peer overlays*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2005.

[12] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):299–314, 2002.

[13] D. Chaum. The dining cryptographers problem: unconditional sender and recipient untraceability. *J. Cryptol.*, 1(1):65–75, 1988.

[14] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, 70(6), 2004.

[15] M. P. Collins and M. K. Reiter. Hit-list worm detection and bot identification in large networks using protocol graphs. In *RAID*, 2007.

[16] M. P. Collins, T. J. Shimeall, S. Faber, J. Janies, R. Weaver, M. De Shon, and J. Kadane. Using uncleanliness to predict future botnet addresses. In *IMC*, pages 93–104, New York, NY, USA, 2007. ACM.

[17] E. Cooke and F. Jahanian. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Steps to Reducing Unwanted Traffic on the Internet Workshop*, 2005.

[18] E. D. Cristofaro and G. Tsudik. Practical private set intersection protocols. Cryptology ePrint Archive, Report 2009/491, 2009. http://eprint.iacr.org/.

[19] D. Dagon, C. Zou, and W. Lee. Modeling botnet propagation using time zones. In *NDSS*, 2006.

[20] I. Damgard and M. Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *Public Key Cryptography*. Springer, 2001.

[21] G. Danezis and P. Mittal. Sybilinfer: Detecting Sybil nodes using social networks. In *NDSS*, 2009.

[22] S. Fortunato, V. Latora, and M. Marchiori. Method to find community structures based on information centrality. *Physical Review E*, 70(5), 2004.

[23] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *ACM conference on Computer and communications security*, pages 375–388, New York, NY, USA, 2007. ACM.

[24] F. C. Freiling, T. Hoz, and G. Wichereski. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *European Symposium on Research in Computer Security*, 2005.

[25] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12), 2002.

[26] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *IEEE INFOCOM*, 2004.

[27] J. Goebel and T. Holz. Rishi: Identify bot contaminated hosts by IRC nickname evaluation. In *HotBots*, 2007.

[28] J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon. Peer-to-peer botnets: Overview and case study. In *HotBots*, 2007.

[29] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of the 17th USENIX Security Symposium (Security'08)*, 2008.

[30] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *USENIX Security Symposium*, 2007.

[31] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[32] K. P. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of ACM SIGCOMM 2003*, Aug. 2003.

[33] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot (NAB): Improving Service Availability in the Face of Botnet Attacks. In *NSDI 2009*, Boston, MA, April 2009.

[34] W. K. Hastings. Monte carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, April

1970.
[35] M. Iliofotou, M. Faloutsos, and M. Mitzenmacher. Exploiting dynamicity in graph-based traffic analysis: Techniques and applications. In *ACM CoNext*, 2009.
[36] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, G. Varghese, and H. Kim. Graption: Automated detection of P2P applications using traffic dispersion graphs (TDGs). In *UC Riverside Technical Report, CS-2008-06080*, 2008.
[37] S. Jarecki and X. Liu. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *Theory of Cryptography Conference*, pages 577–594. Springer, 2009.
[38] M. Jelasity and V. Bilicki. Towards automated detection of peer-to-peer botnets: On the limits of local approaches. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.
[39] M. Jelasity and V. Billicki. Towards automated detection of peer-to-peer botnets: On the limits of local approaches. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.
[40] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying spamming botnets using Botlab. In *NSDI*, 2009.
[41] M. Kaashoek and D. Karger. Koorde: A simple degree-optimal distributed hash table. In *IPTPS*, 2003.
[42] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. Spamalytics: An empirical analysis of spam marketing conversion. In *CCS*, Oct. 2008.
[43] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515, 2004.
[44] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *HotBots*, 2007.
[45] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22:49–86, 1951.
[46] V. Latora and M. Marchiori. Economic small-world behavior in weighted networks. *The European Physical Journal B - Condensed Matter*, 32(2), 2002.
[47] S. Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.
[48] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. In *Proceedings of ACM SIGCOMM*, Aug. 2003.
[49] W. Lu, M. Tavallaee, and A. A. Ghorbani. Automatic discovery of botnet communities on large-scale communication networks. In *ASIACCS*, pages 1–10, New York, NY, USA, 2009. ACM.
[50] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st International Peer To Peer Systems Workshop*, 2002.
[51] S. Nagaraja and R. Anderson. The snooping dragon: social-malware surveillance of the tibetan movement. Technical Report UCAM-CL-TR-746, University of Cambridge, 2009.
[52] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys Rev E Stat Nonlin Soft Matter Phys*, 69(2 Pt 2), 2004.
[53] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74(3), 2006.
[54] D. M. Nicol and N. Schear. Models of privacy preserving traffic tunneling. *Simulation*, 85(9):589–607, 2009.
[55] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt*. Springer, 1999.
[56] D. Pelleg and A. W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 727–734, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
[57] P. Porras, H. Saidi, and V. Yegneswaran. A multi-perspective analysis of the Storm (Peacomm) worm. In *SRI Technical Report 10-01*, 2007.
[58] P. Porras, H. Saidi, and V. Yegneswaran. A foray into Conficker's logic and rendezvous points. In *2nd Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET '09)*, 2009.
[59] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Internet Measurement Conference*, 2006.
[60] A. Ramachandran, D. Dagon, and N. Feamster. Can DNS-based blacklists keep up with bots? In *CEAS*, 2006.
[61] A. Ramachandran, N. Feamster, and D. Dagon. Revealing botnet membership using dnsbl counter-intelligence. In *SRUTI: Proceedings of the 2nd conference on Steps to Reducing Unwanted Traffic on the Internet*, 2006.
[62] D. Randall. Rapidly mixing Markov chains with applications in computer science and physics. *Computing in Science and Engineering*, 8(2):30–41, 2006.
[63] Route views. `http://www.routeviews.org`.
[64] N. Schear and D. M. Nicol. Performance analysis of real traffic carried with encrypted cover flows. In *PADS*, pages 80–87, Washington, DC, USA, 2008. IEEE Computer Society.
[65] S. Sheng, B. Wardman, G. Warner, L. F. Cranor, J. Hong, and C. Zhang. An empirical analysis of phishing blacklists. In *CEAS*, 2009.
[66] A. Sinclair. Improved bounds for mixing rates of markov chains and multicommodity flow. *Combinatorics, Probability and Computing*, 1:351–370, 1992.
[67] E. Spafford and D. Zamboni. Intrusion detection using autonomous agents. *Computer Networks*, 34(4):547–570, 2000.
[68] L. Spitzner. The Honeynet Project: trapping the hackers. *Security & Privacy Magazine, IEEE*, 1(2):15–23, 2003.
[69] E. Stinson and J. C. Mitchell. Characterizing bots' remote control behavior. In *Botnet Detection*. 2008.
[70] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM*, Aug. 2001.
[71] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich. Analysis of the Storm and Nugache trojans: P2P is here. *;login*, 32(6), Dec. 2007.
[72] W. T. Strayer, D. E. Lapsley, R. Walsh, and C. Livadas. Botnet detection based on network behavior. In *Advances in Information Security*. 2008.
[73] G. Varghese and C. Estan. The measurement manifesto. In *HotNets-II*, 2003.
[74] R. Villamarín-Salomón and J. C. Brustoloni. Bayesian bot detection based on dns traffic similarity. In *SAC '09: Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 2035–2041, New York, NY, USA, 2009. ACM.
[75] G. White, E. Fisch, and U. Pooch. Cooperating security managers: a peer-based intrusion detection system. *IEEE Network*, 10(1):20–23, 1996.
[76] T.-F. Yen and M. K. Reiter. Traffic aggregation for malware detection. In *DIMVA '08: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 207–227, Berlin, Heidelberg, 2008. Springer-Verlag.
[77] Q. Zhao, J. Xu, and Z. Liu. Design of a novel statistics counter architecture with optimal space and time efficiency. In *ACM SIGMETRICS*, June 2006.
[78] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. Botgraph: Large scale spamming botnet detection. In *NSDI*, 2009.
[79] M. Zhong, K. Shen, and J. Seiferas. Non-uniform random membership management in peer-to-peer networks. In *INFOCOM*, pages volume 2, 1151–1161, 2005.



# Fast Regular Expression Matching using Small TCAMs for Network Intrusion Detection and Prevention Systems

*Chad R. Meiners Jignesh Patel Eric Norige Eric Torng Alex X. Liu*
*Department of Computer Science and Engineering*
*Michigan State University*
*East Lansing, MI 48824-1226, U.S.A.*
{*meinersc, patelji1, norigeer, torng, alexliu*}*@cse.msu.edu*

## Abstract

Regular expression (RE) matching is a core component of deep packet inspection in modern networking and security devices. In this paper, we propose the first hardware-based RE matching approach that uses Ternary Content Addressable Memories (TCAMs), which are off-the-shelf chips and have been widely deployed in modern networking devices for packet classification. We propose three novel techniques to reduce TCAM space and improve RE matching speed: transition sharing, table consolidation, and variable striding. We tested our techniques on 8 real-world RE sets, and our results show that small TCAMs can be used to store large DFAs and achieve potentially high RE matching throughtput. For space, we were able to store each of the corresponding 8 DFAs with as many as 25,000 states in a 0.59Mb TCAM chip where the number of TCAM bits required per DFA state were 12, 12, 12, 13, 14, 26, 28, and 42. Using a different TCAM encoding scheme that facilitates processing multiple characters per transition, we were able to achieve potential RE matching throughputs of between 10 and 19 Gbps for each of the 8 DFAs using only a single 2.36 Mb TCAM chip.

## 1 Introduction

### 1.1 Background and Problem Statement

Deep packet inspection is a key part of many networking devices on the Internet such as Network Intrusion Detection (or Prevention) Systems (NIDS/NIPS), firewalls, and layer 7 switches. In the past, deep packet inspection typically used *string matching* as a core operator, namely examining whether a packet's payload matches any of a set of predefined strings. Today, deep packet inspection typically uses *regular expression (RE) matching* as a core operator, namely examining whether a packet's payload matches any of a set of predefined regular expressions, because REs are fundamentally more expressive, efficient, and flexible in specifying attack signatures [27]. Most open source and commercial deep packet inspection engines such as Snort, Bro, TippingPoint X505, and many Cisco networking appliances use RE matching. Likewise, some operating systems such as Cisco IOS and Linux have built RE matching into their layer 7 filtering functions. As both traffic rates and signature set sizes are rapidly growing over time, fast and scalable RE matching is now a core network security issue.

RE matching algorithms are typically based on the Deterministic Finite Automata (DFA) representation of regular expressions. A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, A)$, where $Q$ is a set of states, $\Sigma$ is an alphabet, $\delta\colon \Sigma \times Q \to Q$ is the transition function, $q_0$ is the start state, and $A \subseteq Q$ is a set of accepting states. Any set of regular expressions can be converted into an equivalent DFA with the minimum number of states. The fundamental issue with DFA-based algorithms is the large amount of memory required to store transition table $\delta$. We have to store $\delta(q, a) = p$ for each state $q$ and character $a$.

Prior RE matching algorithms are either software-based [4, 6, 7, 12, 16, 18, 19] or FPGA-based [5, 7, 13, 14, 22, 24, 29]. Software-based solutions have to be implemented in customized ASIC chips to achieve high-speed, the limitations of which include high deployment cost and being hard-wired to a specific solution and thus limited ability to adapt to new RE matching solutions. Although FPGA-based solutions can be modified, resynthesizing and updating FPGA circuitry in a deployed system to handle regular expression updates is slow and difficult; this makes FPGA-based solutions difficult to be deployed in many networking devices (such as NIDS/NIPS and firewalls) where the regular expressions need to be updated frequently [18].

### 1.2 Our Approach

To address the limitations of prior art on high-speed RE matching, we propose the first Ternary Content Addressable Memory (TCAM) based RE matching solution. We

use a TCAM and its associated SRAM to encode the transitions of the DFA built from an RE set where one TCAM entry might encode multiple DFA transitions.

TCAM entries and lookup keys are encoded in ternary as 0's, 1's, and *'s where *'s stand for either 0 or 1. A lookup key matches a TCAM entry if and only if the corresponding 0's and 1's match; for example, key 0001101111 matches entry 000110****. TCAM circuits compare a lookup key with all its occupied entries in parallel and return the index (or sometimes the content) of the first address for the content that the key matches; this address is then used to retrieve the corresponding decision in SRAM.

Given an RE set, we first construct an equivalent minimum state DFA [15]. Second, we build a two column TCAM lookup table where each column encodes one of the two inputs to $\delta$: the *source* state ID and the *input* character. Third, for each TCAM entry, we store the *destination* state ID in the same entry of the associated SRAM. Fig. 1 shows an example DFA, its TCAM lookup table, and its SRAM decision table. We illustrate how this DFA processes the input stream "01101111, 01100011". We form a TCAM lookup key by appending the current input character to the current source state ID; in this example, we append the first input character "01101111" to "00", the ID of the initial state $s_0$, to form "0001101111". The first matching entry is the second TCAM entry, so "01", the destination state ID stored in the second SRAM entry is returned. We form the next TCAM lookup key "0101100011" by appending the second input character "011000011" to this returned state ID "01", and the process repeats.

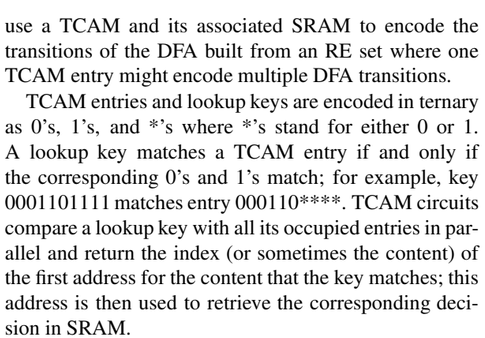



Figure 1: A DFA with its TCAM table

**Advantages of TCAM-based RE Matching** There are three key reasons why TCAM-based RE matching works well. First, *a small TCAM is capable of encoding a large DFA* with carefully designed algorithms leveraging the ternary nature and first-match semantics of TCAMs. Our experimental results show that each of the DFAs built from 8 real-world RE sets with as many as 25,000 states, 4 of which were obtained from the authors of [6], can be stored in a 0.59Mb TCAM chip. The two DFAs that correspond to primarily string matching RE sets require 28 and 42 TCAM bits per DFA state; 5 of the remaining 6 DFAs which have a sizeable number of '.*' patterns require 12 to 14 TCAM bits per DFA state whereas the 6th DFA requires 26 TCAM bits per DFA state. Second, *TCAMs facilitate high-speed RE matching* because TCAMs are essentially high-performance parallel lookup systems: any lookup takes constant time (*i.e.*, a few CPU cycles) regardless of the number of occupied entries. Using Agrawal and Sherwood's TCAM model [1] and the resulting required TCAM sizes for the 8 RE sets, we show that it may be possible to achieve throughputs ranging between 5.36 and 18.6 Gbps using only a single 2.36 Mb TCAM chip. Third, because TCAMs are off-the-shelf chips that are widely deployed in modern networking devices, *it should be easy to design networking devices that include our TCAM based RE matching solution*. It may even be possible to immediately deploy our solution on some existing devices.

**Technical Challenges** There are two key technical challenges in TCAM-based RE matching. The first is encoding a large DFA in a small TCAM. Directly encoding a DFA in a TCAM using one TCAM entry per transition will lead to a prohibitive amount of TCAM space. For example, consider a DFA with 25000 states that consumes one 8 bit character per transition. We would need a total of $140.38\,\text{Mb}\ (= 25000 \times 2^8 \times (8 + \lceil \log 25000 \rceil))$. This is infeasible given the largest available TCAM chip has a capacity of only 72 Mb. To address this challenge, we use two techniques that minimize the TCAM space for storing a DFA: *transition sharing* and *table consolidation*. The second challenge is improving RE matching speed and thus throughput. One way to improve the throughput by up to a factor of $k$ is to use $k$-stride DFAs that consume $k$ input characters per transition. However, this leads to an exponential increase in both state and transition spaces. To avoid this space explosion, we use the novel idea of *variable striding*.

**Key Idea 1 - Transition Sharing** The basic idea is to combine multiple transitions into one TCAM entry by exploiting two properties of DFA transitions: (1) character redundancy where many transitions share the same source state and destination state and differ only in their character label, and (2) state redundancy where many transitions share the same character label and destination state and differ only in their source state. One reason for the pervasive character and state redundancy in DFAs constructed from real-world RE sets is that most states have most of their outgoing transitions going to some common "failure" state; such transitions are often called default transitions. The low entropy of these DFAs opens optimization opportunities. We exploit character redundancy by *character bundling* (*i.e.*, input character sharing) and state redundancy by *shadow encoding* (*i.e.*, source state sharing). In character bundling, we use a ternary encoding of the input character field to represent multiple characters and thus multiple transitions that share the same source and destination states. In shadow encoding, we use a ternary encoding for the source state ID to represent multiple source states and thus multiple transitions that share the same label and destination state.

**Key Idea 2 - Table Consolidation** The basic idea is to merge multiple transition tables into one transition table using the observation that some transition tables share similar structures (*e.g.*, common entries) even if they have different decisions. This shared structure can be exploited by consolidating similar transition tables into one consolidated transition table. When we consolidate $k$ TCAM lookup tables into one consolidated TCAM lookup table, we store $k$ decisions in the associated SRAM decision table.

**Key Idea 3 - Variable Striding** The basic idea is to store transitions with a variety of strides in the TCAM so that we increase the average number of characters consumed per transition while ensuring all the transitions fit within the allocated TCAM space. This idea is based on two key observations. First, for many states, we can capture many but not all $k$-stride transitions using relatively few TCAM entries whereas capturing all $k$-stride transitions requires prohibitively many TCAM entries. Second, with TCAMs, we can store transitions with different strides in the same TCAM lookup table.

The rest of this paper proceeds as follows. We review related work in Section 2. In Sections 3, 4, and 5, we describe transition sharing, table consolidation, and variable striding, respectively. We present implementation issues, experimental results, and conclusions in Sections 6, 7, and 8, respectively.

## 2 Related Work

In the past, deep packet inspection typically used string matching (often called pattern matching) as a core operator; string matching solutions have been extensively studied [2, 3, 28, 30, 32, 33, 35]). TCAM-based solutions have been proposed for string matching, but they do not generalize to RE matching because they only deal with independent strings [3, 30, 35].

Today deep packet inspection often uses RE matching as a core operator because strings are no longer adequate to precisely describe attack signatures [25, 27]. Prior work on RE matching falls into two categories: software-based and FPGA-based. Prior software-based RE matching solutions focus on either reducing memory by minimizing the number of transitions/states or improving speed by increasing the number of characters per lookup. Such solutions can be implemented on general purpose processors, but customized ASIC chip implementations are needed for high speed performance. For transition minimization, two basic approaches have been proposed: alphabet encoding that exploits character redundancy [6, 7, 12, 16] and default transitions that exploit state redundancy [4, 6, 18, 19]. Previous alphabet encoding approaches cannot fully exploit local character redundancy specific to each state. Most use a single alphabet encoding table that can only exploit global character redundancy that applies to every state. Kong *et al.* proposed using 8 alphabet encoding tables by partitioning the DFA states into 8 groups with each group having its own alphabet encoding table [16]. Our work improves upon previous alphabet encoding techniques because we can exploit local character redundancy specific to each state. Our work improves upon the default transition work because we do not need to worry about the number of default transitions that a lookup may go through because TCAMs allow us to traverse an arbitrarily long default transition path in a single lookup. Some transition sharing ideas have been used in some TCAM-based string matching solutions for Aho-Corasick-based DFAs [3, 11]. However, these ideas do not easily extend to DFAs generated by general RE sets, and our techniques produce at least as much transition sharing when restricted to string matching DFAs. For state minimization, two fundamental approaches have been proposed. One approach is to first partition REs into multiple groups and build a DFA from each group; at run time, packet payload needs to be scanned by multiple DFAs [5, 26, 34]. This approach is orthogonal to our work and can be used in combination with our techniques. In particular, because our techniques achieve greater compression of DFAs than previous software-based techniques, less partitioning of REs will be required. The other approach is to use scratch memory to store variables that track the traversal history and avoid some duplication of states [8,17,25]. The benefit of state reduction for scratch memory-based FAs does not come for free. The size of the required scratch memory may be significant, and the time required to update the scratch memory after each transition may be significant. This approach is orthogonal to our approach. While we have only applyied our techniques to DFAs in this initial study of TCAM-based RE matching, our techniques may work very well with scratch memory-based automata.

Prior FPGA-based solutions exploit the parallel processing capabilities of FPGA technology to implement nondeterministic finite automata (NFA) [5, 7, 13, 14, 22, 24,29] or parallel DFAs [23]. While NFAs are more compact than DFAs, they require more memory bandwidth

to process each transition as an NFA may be in multiple states whereas a DFA is always only in one state. Thus, each character that is processed might be processed in up to $|Q|$ transition tables. Prior work has looked at ways for finding good NFA representations of the REs that limit the number of states that need to be processed simultaneously. However, FPGA's cannot be quickly reconfigured, and they have clock speeds that are slower than ASIC chips.

There has been work [7, 12] on creating multi-stride DFAs and NFAs. This work primarily applies to FPGA NFA implementations since multiple character SRAM based DFAs have only been evaluated for a small number of REs. The ability to increase stride has been limited by the constraint that all transitions must be increased in stride; this leads to excessive memory explosion for strides larger than 2. With variable striding, we increase stride selectively on a state by state basis. Alicherry *et al.* have explored variable striding for TCAM-based string matching solutions [3] but not for DFAs that apply to arbitrary RE sets.

# 3 Transition Sharing

The basic idea of transition sharing is to combine multiple transitions into a single TCAM entry. We propose two transition sharing ideas: character bundling and shadow encoding. Character bundling exploits intra-state optimization opportunities and minimizes TCAM tables along the input character dimension. Shadow encoding exploits inter-state optimization opportunities and minimizes TCAM tables along the source state dimension.

## 3.1 Character Bundling

Character bundling exploits character redundancy by combining multiple transitions from the same source state to the same destination into one TCAM entry. Character bundling consists of four steps. (1) Assign each state a unique ID of $\lceil \log |Q| \rceil$ bits. (2) For each state, enumerate all 256 transition rules where for each rule, the predicate is a transition's label and the decision is the destination state ID. (3) For each state, treating the 256 rules as a 1-dimensional packet classifier and leveraging the ternary nature and first-match semantics of TCAMs, we minimize the number of transitions using the optimal 1-dimensional TCAM minimization algorithm in [20, 31]. (4) Concatenate the $|Q|$ 1-dimensional minimal prefix classifiers together by prepending each rule with its source state ID. The resulting list can be viewed as a 2-dimensional classifier where the two fields are source state ID and transition label and the decision is the destination state ID. Fig. 1 shows an example DFA and its TCAM lookup table built using character bundling. The three chunks of TCAM entries encode the 256 transitions for $s_0$, $s_1$, and $s_2$, respectively. Without character bundling, we would need $256 \times 3$ entries.

## 3.2 Shadow Encoding

Whereas character bundling uses ternary codes in the input character field to encode multiple input characters, shadow encoding uses ternary codes in the source state ID field to encode multiple source states.

### 3.2.1 Observations

We use our running example in Fig. 1 to illustrate shadow encoding. We observe that all transitions with source states $s_1$ and $s_2$ have the same destination state except for the transitions on character $c$. Likewise, source state $s_0$ differs from source states $s_1$ and $s_2$ only in the character range $[a, o]$. This implies there is a lot of state redundancy. The table in Fig. 2 shows how we can exploit state redundancy to further reduce required TCAM space. First, since states $s_1$ and $s_2$ are more similar, we give them the state IDs 00 and 01, respectively. State $s_2$ uses the ternary code of 0* in the state ID field of its TCAM entries to share transitions with state $s_1$. We give state $s_0$ the state ID of 10, and it uses the ternary code of $**$ in the state ID field of its TCAM entries to share transitions with both states $s_1$ and $s_2$. Second, we order the state tables in the TCAM so that state $s_1$ is first, state $s_2$ is second, and state $s_0$ is last. This facilitates the sharing of transitions among different states where earlier states have incomplete tables deferring some transitions to later tables.

| TCAM | | | SRAM |
|---|---|---|---|
| Src State | ID | Input | Dest State ID |
| $s_1$ | 00 | 0110 0011 | 01: $s_2$ |
| | 0* | 0110 001* | 00: $s_1$ |
| $s_2$ | 0* | 0110 0000 | 10: $s_0$ |
| | 0* | 0110 **** | 01: $s_2$ |
| | ** | 0110 0000 | 10: $s_0$ |
| $s_0$ | ** | 0110 **** | 00: $s_1$ |
| | ** | **** **** | 10: $s_0$ |

Figure 2: TCAM table with shadow encoding

We must solve three problems to implement shadow encoding: (1) Find the best order of the state tables in the TCAM given that any order is allowed. (2) Identify entries to remove from each state table given this order. (3) Choose binary IDs and ternary codes for each state that support the given order and removed entries. We solve these problems in the rest of this section.

Our shadow encoding technique builds upon prior work with default transitions [4, 6, 18, 19] by exploiting the same state redundancy observation and using their concepts of default transitions and Delayed input DFAs (D$^2$FA). However, our final technical solutions are different because we work with TCAM whereas prior techniques work with RAM. For example, the concept of a ternary state code has no meaning when working with RAM. The key advantage of shadow encoding in TCAM over prior default transition techniques is speed. Specifically, shadow encoding incurs no delay while prior default transition techniques incur significant delay because a DFA may have to traverse multiple default transitions before consuming an input character.

### 3.2.2 Determining Table Order

We first describe how we compute the order of tables within the TCAM. We use some concepts such as default transitions and D$^2$FA that were originally defined by Kumar *et al.* [18] and subsequently refined [4, 6, 19].

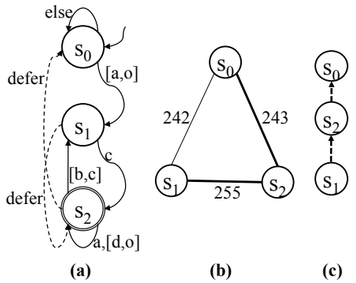



Figure 3: D$^2$FA, SRG, and deferment tree

A D$^2$FA is a DFA with default transitions where each state $p$ can have at most one default transition to one other state $q$ in the D$^2$FA. In a legal D$^2$FA, the directed graph consisting of only default transitions must be acyclic; we call this graph a *deferment forest*. It is a forest rather than a tree since more than one node may not have a default transition. We call a tree in a deferment forest a *deferment tree*.

We determine the order of state tables in TCAM by constructing a deferment forest and then using the partial order defined by the deferment forest. Specifically, if there is a directed path from state $p$ to state $q$ in the deferment forest, we say that state $p$ *defers* to state $q$, denoted $p \succ q$. If $p \succ q$, we say that state $p$ is in state $q$'s shadow. We use the partial order of a deferment forest to determine the order of state transition tables in the TCAM. Specifically, state $q$'s transition table must be placed after the transition tables of all states in state $q$'s shadow.

We compute a deferment forest that minimizes the TCAM representation of the resulting D$^2$FA as follows. Our algorithm builds upon algorithms from prior work [4, 6, 18, 19], but there are several key differences. First, unlike prior work, we do not pay a speed penalty for long default transition paths. Thus, we achieve better transition sharing than prior work. Second, to maximize the potential gains from our variable striding technique described in Section 5 and table consolidation, we choose states that have lots of self-loops to be the roots of our deferment trees. Prior work has typically chosen roots in order to minimize the distance from a leaf node to a root, though Becchi and Crowley do consider related criteria when constructing their D$^2$FA [6]. Third, we explicitly ignore transition sharing between states that have few transitions in common. This has been done implicitly in the past, but we show how doing so leads to better results when we use table consolidation.

The algorithm for constructing deferment forests consists of four steps. First, we construct a Space Reduction Graph (SRG), which was proposed in [18], from a given DFA. Given a DFA with $|Q|$ states, an SRG is a clique with $|Q|$ vertices each representing a distinct state. The weight of each edge is the number of common (outgoing) transitions between the two connected states. Second, we trim away edges with small weight from the SRG. In our experiments, we use a cutoff of 10. We justify this step based on the following observations. A key property of SRGs that we observed in our experiments is that the weight distribution is bimodal: an edge weight is typically either very small ($< 10$) or very large ($> 180$). If we use these low weight edges for default transitions, the resulting TCAM often has more entries. Plus, we get fewer deferment trees which hinders our table consolidation technique (Section 4). Third, we compute a deferment forest by running Kruskal's algorithm to find a maximum weight spanning forest. Fourth, for each deferment tree, we pick the state that has largest number of transitions going back to itself as the root. Fig. 3(b) and (c) show the SRG and the deferment tree, respectively, for the DFA in Fig. 1.

We make the following key observation about the root states in our deferment trees. In most deferment trees, more than 128 (*i.e.*, half) of the root state's outgoing transitions lead back to the root state; we call such a state a *self-looping state*. Based on the pigeonhole principle and the observed bimodal distribution, each deferment tree can have at most one self-looping state, and it is clearly the root state. We choose self-looping states as roots to improve the effectiveness of variable striding which we describe in Section 5. Intuitively, we have a very space-efficient method, self-loop unrolling, for increasing the stride of self-looping root states. The resulting increase in stride applies to all states that defer transitions to this self-looping root state.

When we apply Kruskal's algorithm, we use a tie breaking strategy because many edges have the same weight. To have most deferment trees centered around a self-looping state, we give priority to edges that have the self-looping state as one endpoint. If we still have a

tie, we favor edges by the total number of edges in the current spanning tree that both endpoints are connected to prioritize nodes that are already well connected.

### 3.2.3 Choosing Transitions

For a given DFA and a corresponding deferment forest, we construct a D$^2$FA as follows. If state $p$ has a default transition to state $q$, we remove any transitions that are common to both $p$'s transition table and $q$'s transition table from $p$'s transition table. We denote the default transition in the D$^2$FA with a dashed arrow labeled with defer. Fig. 3(a) shows the D$^2$FA for the DFA in Fig. 1 given the corresponding deferment forest (a deferment tree in this case) in Figure 3(c). We now compute the TCAM entries for each transition table.

(1) For each state, enumerate all individual transition rules except the deferred transitions. For each transition rule, the predicate is the label of the transition and the decision is the *state ID* of the destination state. For now, we just ensure each state has a unique state ID. Thus, we get an incomplete 1-dimensional classifier for each state. (2) For each state, we minimize its transition table using the 1-dimensional incomplete classifier minimization algorithm in [21]. This algorithm works by first adding a default rule with a unique decision that has weight larger than the size of the domain, then applying the weighted one-dimensional TCAM minimization algorithm in [20] to the resulting complete classifier, and finally remove the default rule, which is guaranteed to remain the default rule in the minimal complete classifier due to its huge weight. In our solution, the character bundling technique is used in this step. We also consider some optimizations where we specify some deferred transitions to reduce the total number of TCAM entries. For example, the second entry in $s_2$'s table in Fig. 2 is actually a deferred transition to state $s_0$'s table, but not using it would result in 4 TCAM entries to specify the transitions that $s_2$ does not share with $s_0$.

### 3.2.4 Shadow Encoding Algorithm

To ensure that proper sharing of transitions occurs, we need to encode the source state IDs of the TCAM entries according to the following shadow encoding scheme. Each state is assigned a binary *state ID* and a ternary *shadow code*. State IDs are used in the decisions of transition rules. Shadow codes are used in the source state ID field of transition rules. In a valid assignment, every state ID and shadow code must have the same number of bits, which we call the *shadow length* of the assignment. For each state $p$, we use $ID(p)$ and $SC(p)$ to denote the state ID and shadow code of $p$. A valid assignment of state IDs and shadow codes for a deferment forest must satisfy the following four shadow encoding properties:

1. *Uniqueness Property*: For any two distinct states $p$ and $q$, $ID(p) \neq ID(q)$ and $SC(p) \neq SC(q)$.
2. *Self-Matching Property*: For any state $p$, $ID(p) \in SC(p)$ (*i.e.*, $ID(p)$ matches $SC(p)$).
3. *Deferment Property*: For any two states $p$ and $q$, $p \succ q$ (*i.e.*, $q$ is an ancestor of $p$ in the given deferment tree) if and only if $SC(p) \subset SC(q)$.
4. *Non-interception Property*: For any two distinct states $p$ and $q$, $p \succ q$ if and only if $ID(p) \in SC(q)$.

Intuitively, $q$'s shadow code must include the state ID of all states in $q$'s shadow and cannot include the state ID of any states not in $q$'s shadow.

We give an algorithm for computing a valid assignment of state IDs and shadow codes for each state given a single deferment tree $DT$. We handle deferment forests by simply creating a virtual root node whose children are the roots of the deferment trees in the forest and then running the algorithm on this tree. In the following, we refer to states as nodes.

Our algorithm uses the following internal variables for each node $v$: a local binary ID denoted $L(v)$, a global binary ID denoted $G(v)$, and an integer weight denoted $W(v)$ that is the shadow length we would use for the subtree of $DT$ rooted at $v$. Intuitively, the state ID of $v$ will be $G(v)|L(v)$ where $|$ denotes concatenation, and the shadow code of $v$ will be the prefix string $G(v)$ followed by the required number of *'s; some extra padding characters may be needed. We use $\#L(v)$ and $\#G(v)$to denote the number of bits in $L(v)$ and $G(v)$, respectively.

Our algorithm processes nodes in a bottom-up fashion. For each node $v$, we initially set $L(v) = G(v) = \emptyset$ and $W(v) = 0$. Each leaf node of $DT$ is now processed, which we denote by marking them red. We process an internal node $v$ when all its children $v_1, \cdots, v_n$ are red. Once a node $v$ is processed, its weight $W(v)$ and its local ID $L(v)$ are fixed, but we will prepend additional bits to its global ID $G(v)$ when we process its ancestors in $DT$.

We assign $v$ and each of its children a variable-length binary code, which we call *HCode*. The HCode provides a unique signature that uniquely distinguishes each of the $n+1$ nodes from each other while satisfying the four required shadow code properties. One option would be to simply use $\lg(n+1)$ bits and assign each node a binary number from 0 to $n$. However, to minimize the shadow code length $W(v)$, we use a Huffman coding style algorithm instead to compute the HCodes and $W(v)$. This algorithm uses two data structures: a binary encoding tree $T$ with $n+1$ leaf nodes, one for $v$ and each of its children, and a min-priority queue, initialized with $n+1$ elements, one for $v$ and each of its children, that is ordered by node weight. While the priority queue has more than one element, we remove the two elements $x$ and $y$ with lowest weight from the priority queue, create a new



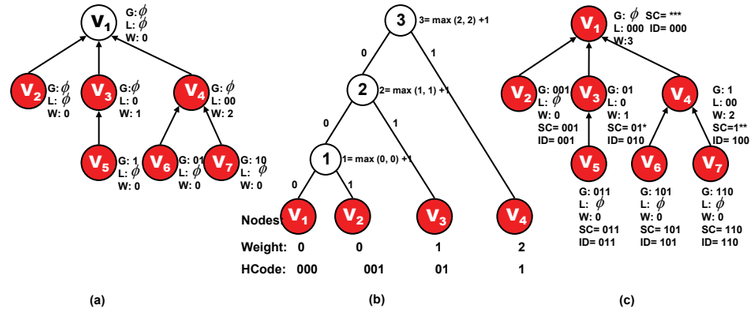



Figure 4: Shadow encoding example

internal node $z$ in $T$ with two children $x$ and $y$ and set weight(z)=maximum(weight(x), weight(y))+1, and then put element $z$ into the priority queue. When there is only a single element in the priority queue, the binary encoding tree $T$ is complete. The HCode assigned to each leaf node $v'$ is the path in $T$ from the root node to $v'$ where left edges have value 0 and right edges have value 1. We update the internal variables of $v$ and its descendants in $DT$ as follows. We set $L(v)$ to be its HCode, and $W(v)$ to be the weight of the root node of $T$; $G(v)$ is left empty. For each child $v_i$, we prepend $v_i$'s HCode to the global ID of every node in the subtree rooted at $v_i$ including $v_i$ itself. We then mark $v$ as red. This continues until all nodes are red.

We now assign each node a state ID and a shadow code. First, we set the shadow length to be $k$, the weight of the root node of $DT$. We use $\{*\}^m$ to denote a ternary string with $m$ number of *'s and $\{0\}^m$ to denote a binary string with $m$ number of 0's. For each node $v$, we compute $v$'s state ID and shadow code as follows: $ID(v) = G(v)|L(v)|\{0\}^{k-\#G(v)-\#L(v)}, SC(v) = G(v)|\{*\}^{k-\#G(v)}$. We illustrate our shadow encoding algorithm in Figure 4. Figure 4(a) shows all the internal variables just before $v_1$ is processed. Figure 4(b) shows the Huffman style binary encoding tree $T$ built for node $v_1$ and its children $v_2$, $v_3$, and $v_4$ and the resulting HCodes. Figure 4(c) shows each node's final weight, global ID, local ID, state ID and shadow code.

Experimentally, we found that our shadow encoding algorithm is effective at minimizing shadow length. No DFA had a shadow length larger than $\lceil \log_2 |Q| \rceil + 3$, and $\lceil \log_2 |Q| \rceil$ is the minimum possible shadow length.

## 4 Table Consolidation

We now present *table consolidation* where we combine multiple transition tables for different states into a single transition table such that the combined table takes less TCAM space than the total TCAM space used by the original tables. To define table consolidation, we need two new concepts: $k$-decision rule and $k$-decision table. A *$k$-decision rule* is a rule whose decision is an array of $k$ decisions. A *$k$-decision table* is a sequence of $k$-decision rules following the first-match semantics. Given a $k$-decision table $\mathbb{T}$ and $i$ ($0 \leq i < k$), if for any rule $r$ in $\mathbb{T}$ we delete all the decisions except the $i$-th decision, we get a 1-decision table, which we denote as $\mathbb{T}[i]$. In table consolidation, we take a set of $k$ 1-decision tables $\mathbb{T}_0, \cdots, \mathbb{T}_{k-1}$ and construct a $k$-decision table $\mathbb{T}$ such that for any $i$ ($0 \leq i < k$), the condition $\mathbb{T}_i \equiv \mathbb{T}[i]$ holds where $\mathbb{T}_i \equiv \mathbb{T}[i]$ means that $\mathbb{T}_i$ and $\mathbb{T}[i]$ are equivalent (*i.e.*, they have the same decision for every search key). We call the process of computing $k$-decision table $\mathbb{T}$ *table consolidation*, and we call $\mathbb{T}$ the *consolidated table*.

### 4.1 Observations

Table consolidation is based three observations. First, semantically different TCAM tables may share common entries with possibly different decisions. For example, the three tables for $s_0$, $s_1$ and $s_2$ in Fig. 1 have three entries in common: 01100000, 0110****, and ********. Table consolidation provides a novel way to remove such information redundancy. Second, given any set of $k$ 1-decision tables $\mathbb{T}_0, \cdots, \mathbb{T}_{k-1}$, we can always find a $k$-decision table $\mathbb{T}$ such that for any $i$ ($0 \leq i < k$), the condition $\mathbb{T}_i \equiv \mathbb{T}[i]$ holds. This is easy to prove as we can use one entry per each possible binary search key in $\mathbb{T}$. Third, a TCAM chip typically has a build-in SRAM module that is commonly used to store lookup decisions. For a TCAM with $n$ entries, the SRAM module is arranged as an array of $n$ entries where SRAM[i] stores the decision of TCAM[i] for every $i$. A TCAM lookup returns the index of the first matching entry in the TCAM, which is then used as the index to directly find the corresponding decision in the SRAM. In table consolidation, we essentially trade SRAM space for TCAM space because each SRAM entry needs to store multiple decisions. As SRAM is cheaper and more efficient than

TCAM, moderately increasing SRAM usage to decrease TCAM usage is worthwhile.

Fig. 5 shows the TCAM lookup table and the SRAM decision table for a 3-decision consolidated table for states $s_0$, $s_1$, and $s_2$ in Fig. 1. In this example, by table consolidation, we reduce the number of TCAM entries from 11 to 5 for storing the transition tables for states $s_0$, $s_1$, and $s_2$. This consolidated table has an ID of 0. As both the table ID and column ID are needed to encode a state, we use the notation $< Table\ ID > @ < Column\ ID >$ to represent a state.

TCAM

| Consolidated Src Table ID | Input Character |
|---|---|
| 0 | 0110 0000 |
| 0 | 0110 0010 |
| 0 | 0110 0011 |
| 0 | 0110 **** |
| 0 | **** **** |

SRAM

| Column ID | | |
|---|---|---|
| 00 | 01 | 10 |
| $s_0$ | $s_0$ | $s_0$ |
| $s_1$ | $s_1$ | $s_1$ |
| $s_1$ | $s_2$ | $s_1$ |
| $s_1$ | $s_2$ | $s_2$ |
| $s_0$ | $s_0$ | $s_0$ |

Figure 5: 3-decision table for 3 states in Fig. 1

There are two key technical challenges in table consolidation. The first challenge is how to consolidate $k$ 1-decision transition tables into a $k$-decision transition table. The second challenge is which 1-decision transition tables should be consolidated together. Intuitively, the more similar two 1-decision transition tables are, the more TCAM space saving we can get from consolidating them together. However, we have to consider the deferment relationship among states. We present our solutions to these two challenges.

## 4.2 Computing a $k$-decision table

In this section, we assume we know which states need to be consolidated together and present a local state consolidation algorithm that takes a $k_1$-decision table for state set $S_i$ and a $k_2$-decision table for another state set $S_j$ as its input and outputs a consolidated $(k_1 + k_2)$-decision table for state set $S_i \cup S_j$. For ease of presentation, we first assume that $k_1 = k_2 = 1$.

Let $s_1$ and $s_2$ be the two input states which have default transitions to states $s_3$ and $s_4$. We enforce a constraint that if we do not consolidate $s_3$ and $s_4$ together, then $s_1$ and $s_2$ cannot defer any transitions at all. If we do consolidate $s_3$ and $s_4$ together, then $s_1$ and $s_2$ may have incomplete transition tables due to default transitions to $s_3$ and $s_4$, respectively. We assign state $s_1$ column ID 0 and state $s_2$ column ID 1. This consolidated table will be assigned a common table ID $X$. Thus, we encode $s_1$ as $X@0$ and $s_2$ as $X@1$.

The key concepts underlying this algorithm are breakpoints and critical ranges. To define breakpoints, it is helpful to view $\Sigma$ as numbers ranging from 0 to $|\Sigma| - 1$; given 8 bit characters, $|\Sigma| = 256$. For any state $s$, we define a character $i \in \Sigma$ to be a *breakpoint* for $s$ if $\delta(s, i) \neq \delta(s, i-1)$. For the end cases, we define 0 and $|\Sigma|$ to be breakpoints for every state $s$. Let $b(s)$ be the set of breakpoints for state $s$. We then define $b(S) = \bigcup_{s \in S} b(s)$ to be the set of breakpoints for a set of states $S \subset Q$. Finally, for any set of states $S$, we define $r(S)$ to be the set of ranges defined by $b(S)$: $r(S) = \{[0, b_2 - 1], [b_2, b_3 - 1], \ldots, [b_{|b(S)|-1}, |\Sigma| - 1]\}$ where $b_i$ is $i$th smallest breakpoint in $b(S)$. Note that $0 = b_1$ is the smallest breakpoint and $|\Sigma|$ is the largest breakpoint in $b(S)$. Within $r(S)$, we label the range beginning at breakpoint $b_i$ as $r_i$ for $1 \leq i \leq |b(S)| - 1$. If $\delta(s, b_i)$ is deferred, then $r_i$ is a deferred range.

When we consolidate $s_1$ and $s_2$ together, we compute $b(\{s_1, s_2\})$ and $r(\{s_1, s_2\})$. For each $r' \in r(\{s_1, s_2\})$ where $r'$ is not a deferred range for both $s_1$ and $s_2$, we create a consolidated transition rule where the decision of the entry is the ordered pair of decisions for state $s_1$ and $s_2$ on $r'$. For each $r' \in r(\{s_1, s_2\})$ where $r'$ is a deferred range for one of $s_1$ but not the other, we fill in $r'$ in the incomplete transition table where it is deferred, and we create a consolidated entry where the decision of the entry is the ordered pair of decisions for state $s_1$ and $s_2$ on $r'$. Finally, for each $r' \in r(\{s_1, s_2\})$ where $r'$ is a deferred range for both $s_1$ and $s_2$, we do not create a consolidated entry. This produces a non-overlapping set of transition rules that may be incomplete if some ranges do not have a consolidated entry. If the final consolidated transition table is complete, we minimize it using the optimal 1-dimensional TCAM minimization algorithm in [20, 31]. If the table is incomplete, we minimize it using the 1-dimensional incomplete classifier minimization algorithm in [21]. We generalize this algorithm to cases where $k_1 > 1$ and $k_2 > 1$ by simply considering $k_1 + k_2$ states when computing breakpoints and ranges.

## 4.3 Choosing States to Consolidate

We now describe our global consolidation algorithm for determining which states to consolidate together. As we observed earlier, if we want to consolidate two states $s_1$ and $s_2$ together, we need to consolidate their parent nodes in the deferment forest as well or else lose all the benefits of shadow encoding. Thus, we propose to consolidate two deferment trees together.

A consolidated deferment tree must satisfy the following properties. First, each node is to be consolidated with at most one node in the second tree; some nodes may not be consolidated with any node in the second tree. Second, a level $i$ node in one tree must be consolidated with a level $i$ node in the second tree. The level of a node is its distance from the root. We define the root to be a level 0 node. Third, if two level $i$ nodes are consolidated together, their level $i - 1$ parent nodes must also be consolidated together. An example legal matching of nodes between two deferment trees is depicted in Fig. 6.

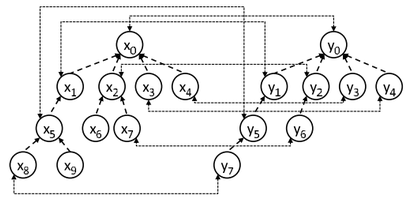



Figure 6: Consolidating two trees

Given two deferment trees, we start the consolidation process from the roots. After we consolidate the two roots, we need to decide how to pair their children together. For each pair of nodes that are consolidated together, we again must choose how to pair their children together, and so on. We make an optimal choice using a combination of dynamic programming and matching techniques. Our algorithm proceeds as follows. Suppose we wish to compute the minimum cost $C(x, y)$, measured in TCAM entries, of consolidating two subtrees rooted at nodes $x$ and $y$ where $x$ has $u$ children $X = \{x_1, \ldots, x_u\}$ and $y$ has $v$ children $Y = \{y_1, \ldots, y_v\}$. We first recursively compute $C(x_i, y_j)$ for $1 \leq i \leq u$ and $1 \leq j \leq v$ using our local state consolidation algorithm as a subroutine. We then construct a complete bipartite graph $K_{X,Y}$ such that each edge $(x_i, y_j)$ has the edge weight $C(x_i, y_j)$ for $1 \leq i \leq u$ and $1 \leq j \leq v$. Here $C(x, y)$ is the cost of a minimum weight matching of $K(X, Y)$ plus the cost of consolidating $x$ and $y$. When $|X| \neq |Y|$, to make the sets equal in size, we pad the smaller set with null states that defer all transitions.

Finally, we must decide which trees to consolidate together. We assume that we produce $k$-decision tables where $k$ is a power of 2. We describe how we solve the problem for $k = 2$ first. We create an edge-weighted complete graph with where each deferment tree is a node and where the weight of each edge is the cost of consolidating the two corresponding deferment trees together. We find a minimum weight matching of this complete graph to give us an optimal pairing for $k = 2$. For larger $k = 2^l$, we then repeat this process $l - 1$ times. Our matching is not necessarily optimal for $k > 2$.

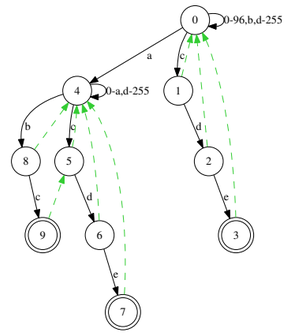



Figure 7: D$^2$FA for {a.*bc, cde}

In some cases, the deferment forest may have only one tree. In such cases, we consider consolidating the subtrees rooted at the children of the root of the single deferment tree. We also consider similar options if we have a few deferment trees but they are not structurally similar.

## 4.4 Effectiveness of Table Consolidation

We now explain why table consolidation works well on real-world RE sets. Most real-world RE sets contain REs with wildcard closures '.*' where the wildcard '.' matches any character and the closure '*' allows for unlimited repetitions of the preceding character. Wildcard closures create deferment trees with lots of structural similarity. For example, consider the D$^2$FA in Fig. 7 for RE set `\{a.*bc, cde\}` where we use dashed arrows to represent the default transitions. The wildcard closure '.*' in the RE `a.*bc` duplicates the entire DFA sub-structure for recognizing string `cde`. Thus, table consolidation of the subtree $(0, 1, 2, 3)$ with the subtree $(4, 5, 6, 7)$ will lead to significant space saving.

# 5 Variable Striding

We explore ways to improve RE matching throughput by consuming multiple characters per TCAM lookup. One possibility is a $k$-stride DFA which uses $k$-stride transitions that consume $k$ characters per transition. Although $k$-stride DFAs can speed up RE matching by up to a factor of $k$, the number of states and transitions can grow exponentially in $k$. To limit the state and transition space explosion, we propose variable striding using *variable-stride DFAs*. A $k$-var-stride DFA consumes between 1 and $k$ characters in each transition with at least one transition consuming $k$ characters. Conceptually, each state in a $k$-var-stride DFA has $256^k$ transitions, and each transition is labeled with (1) a unique string of $k$ characters and (2) a stride length $j$ ($1 \leq j \leq k$) indicating the number of characters consumed.

In TCAM-based variable striding, each TCAM lookup uses the next $k$ consecutive characters as the lookup key, but the number of characters consumed in the lookup varies from 1 to $k$; thus, the lookup decision contains both the destination state ID and the stride length.

## 5.1 Observations

We use an example to show how variable striding can achieve a significant RE matching throughput increase with a small and controllable space increase. Fig. 8 shows a 3-var-stride transition table that corresponds to state $s_0$ in Figure 1. This table only has 7 entries as opposed to 116 entries in a full 3-stride table for $s_0$. If we assume that each of the 256 characters is equally likely to occur, the average number of characters consumed per

3-var-stride transition of $s_0$ is $1 * 1/16 + 2 * 15/256 + 3 * 225/256 = 2.82$.

TCAM

| SRC | Input |
|---|---|
| $s_0$ | 0110 0000 **** **** **** **** |
| $s_0$ | 0110 **** **** **** **** **** |
| $s_0$ | **** **** 0110 0000 **** **** |
| $s_0$ | **** **** 0110 **** **** **** |
| $s_0$ | **** **** **** **** 0110 0000 |
| $s_0$ | **** **** **** **** 0110 **** |
| $s_0$ | **** **** **** **** **** **** |

SRAM

| DEC : Stride |
|---|
| $s_0$ : 1 |
| $s_1$ : 1 |
| $s_0$ : 2 |
| $s_1$ : 2 |
| $s_0$ : 3 |
| $s_1$ : 3 |
| $s_0$ : 3 |

Figure 8: 3-var-stride transition table for $s_0$

## 5.2 Eliminating State Explosion

We first explain how converting a 1-stride DFA to a $k$-stride DFA causes state explosion. For a source state and a destination state pair $(s, d)$, a $k$-stride transition path from $s$ to $d$ may contain $k-1$ intermediate states (excluding $d$); for each unique combination of accepting states that appear on a $k$-stride transition path from $s$ to $d$, we need to create a new destination state because a unique combination of accepting states implies that the input has matched a unique combination of REs. This can be a very large number of new states.

We eliminate state explosion by ending any $k$-var-stride transition path at the first accepting state it reaches. Thus, a $k$-var-stride DFA has the exact same state set as its corresponding 1-stride DFA. Ending $k$-var-stride transitions at accepting states does have subtle interactions with table consolidation and shadow encoding. We end any $k$-var-stride consolidated transition path at the first accepting state reached in any one of the paths being consolidated which can reduce the expected throughput increase of variable striding. There is a similar but even more subtle interaction with shadow encoding which we describe in the next section.

## 5.3 Controlling Transition Explosion

In a $k$-stride DFA converted from a 1-stride DFA with alphabet $\Sigma$, a state has $|\Sigma|^k$ outgoing $k$-stride transitions. Although we can leverage our techniques of character bundling and shadow encoding to minimize the number of required TCAM entries, the rate of growth tends to be exponential with respect to stride length $k$. We have two key ideas to control transition explosion: $k$-var-stride transition sharing and self-loop unrolling.

### 5.3.1 $k$-var-stride Transition Sharing Algorithm

Similar to 1-stride DFAs, there are many transition sharing opportunities in a $k$-var-stride DFA. Consider two states $s_0$ and $s_1$ in a 1-stride DFA where $s_0$ defers to $s_1$. The deferment relationship implies that $s_0$ shares many common 1-stride transitions with $s_1$. In the $k$-var-stride DFA constructed from the 1-stride DFA, all $k$-var-stride transitions that begin with these common 1-stride transitions are also shared between $s_0$ and $s_1$. Furthermore, two transitions that do not begin with these common 1-stride transitions may still be shared between $s_0$ and $s_1$. For example, in the 1-stride DFA fragment in Fig. 9, although $s_1$ and $s_2$ do not share a common transition for character $a$, when we construct the 2-var-stride DFA, $s_1$ and $s_2$ share the same 2-stride transition on string $aa$ that ends at state $s_5$.

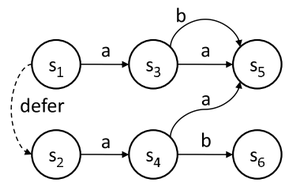



Figure 9: $s_1$ and $s_2$ share transition $aa$

To promote transition sharing among states in a $k$-var-stride DFA, we first need to decide on the deferment relationship among states. The ideal deferment relationship should be calculated based on the SRG of the final $k$-var-stride DFA. However, the $k$-var-stride DFA cannot be finalized before we need to compute the deferment relationship among states because the final $k$-var-stride DFA is subject to many factors such as available TCAM space. There are two approximation options for the final $k$-var-stride DFA for calculating the deferment relationship: the 1-stride DFA and the full $k$-stride DFA. We have tried both options in our experiments, and the difference in the resulting TCAM space is negligible. Thus, we simply use the deferment forest of the 1-stride DFA in computing the transition tables for the $k$-var-stride DFA.

Second, for any two states $s_1$ and $s_2$ where $s_1$ defers to $s_2$, we need to compute $s_1$'s $k$-var-stride transitions that are not shared with $s_2$ because those transitions will constitute $s_1$'s $k$-var-stride transition table. Although this computation is trivial for 1-stride DFAs, this is a significant challenge for $k$-var-stride DFAs because each state has too many $(256^k)$ $k$-var-stride transitions. The straightforward algorithm that enumerates all transitions has a time complexity of $O(|Q|^2|\Sigma|^k)$, which grows exponentially with $k$. We propose a dynamic programming algorithm with a time complexity of $O(|Q|^2|\Sigma|k)$, which grows linearly with $k$. Our key idea is that the non-shared transitions for a $k$-stride DFA can be quickly computed from the non-shared transitions of a (k-1)-var-stride DFA. For example, consider the two states $s_1$ and $s_2$ in Fig. 9 where $s_1$ defers to $s_2$. For character $a$, $s_1$ transits to $s_3$ while $s_2$ transits to $s_4$. Assuming that we have computed all (k-1)-var-stride transitions of $s_3$ that are not shared with the (k-1)-var-stride transitions of $s_4$, if we prepend all these (k-1)-var-stride transitions with character $a$, the resulting $k$-var-stride transitions of $s_1$ are all not shared with the $k$-var-stride transitions of $s_2$, and therefore should all be included in $s_1$'s $k$-var-stride transition table. Formally, using $n(s_i, s_j, k)$ to denote the number of $k$-stride transitions of $s_i$ that are not shared with $s_j$, our dynamic programming algorithm uses the following recursive relationship between $n(s_i, s_j, k)$ and $n(s_i, s_j, k-1)$:

$$n(s_i, s_j, 0) = \begin{cases} 0 & \text{if } s_i = s_j \\ 1 & \text{if } s_i \neq s_j \end{cases} \quad (1)$$

$$n(s_i, s_j, k) = \sum_{c \in \Sigma} n(\delta(s_i, c), \delta(s_j, c), k-1) \quad (2)$$

The above formulae assume that the intermediate states on the $k$-stride paths starting from $s_i$ or $s_j$ are all non-accepting. For state $s_i$, we stop increasing the stride length along a path whenever we encounter an accepting state on that path or on the corresponding path starting from $s_j$. The reason is similar to why we stop a consolidated path at an accepting state, but the reasoning is more subtle.

Let $p$ be the string that leads $s_j$ to an accepting state. The key observation is that we know that any $k$-var-stride path that starts from $s_j$ and begins with $p$ ends at that accepting state. This means that $s_i$ cannot exploit transition sharing on any strings that begin with $p$.

The above dynamic programming algorithm produces non-overlapping and and incomplete transition tables that we compress using the 1-dimensional incomplete classifier minimization algorithm in [21].

### 5.3.2 Self-Loop Unrolling Algorithm

We now consider root states, most of which are self-looping. We have two methods to compute the $k$-var-stride transition tables of root states. The first is direct expansion (stopping transitions at accepting states) since these states do not defer to other states which results in an exponential increase in table size with respect to $k$. The second method, which we call *self-loop unrolling*, scales linearly with $k$.

Self-loop unrolling increases the stride of all the self-loop transitions encoded by the last default TCAM entry. Self-loop unrolling starts with a root state $j$-var-stride transition table encoded as a compressed TCAM table of $n$ entries with a final default entry representing most of the self-loops of the root state. Note that given any complete TCAM table where the last entry is not a default entry, we can always replace that last entry with a default entry without changing the semantics of the table. We generate the (j+1)-var-stride transition table by expanding the last default entry into $n$ new entries, which are obtained by prepending 8 *s as an extra default field to the beginning of the original $n$ entries. This produces a (j+1)-var-stride transition table with $2n - 1$ entries. Fig. 8 shows the resulting table when we apply self-loop unrolling twice on the DFA in Fig. 1.

## 5.4 Variable Striding Selection Algorithm

We now propose solutions for the third key challenge - which states should have their stride lengths increased and by how much, *i.e.*, how should we compute the transition function $\delta$. Note that each state can independently choose its variable striding length as long as the final transition tables are composed together according to the deferment forest. This can be easily proven based on the way that we generate $k$-var-stride transition tables. For any two states $s_1$ and $s_2$ where $s_1$ defers to $s_2$, the way that we generate $s_1$'s $k$-var-stride transition table is seemingly based on the assumption that $s_2$'s transition table is also $k$-var-stride; actually, we do not have this assumption. For example, if we choose $k$-var-stride $(2 \leq k)$ for $s_1$ and 1-stride for $s_2$, all strings from $s_1$ will be processed correctly; the only issue is that strings deferred to $s_2$ will process only one character.

We view this as a packing problem: given a TCAM capacity $C$, for each state $s$, we select a variable stride length value $K_s$, such that $\sum_{s \in Q} |\mathbb{T}(s, K_s)| \leq C$, where $\mathbb{T}(s, K_s)$ denotes the $K_s$-var-stride transition table of state $s$. This packing problem has a flavor of the knapsack problem, but an exact formulation of an optimization function is impossible without making assumptions about the input character distribution. We propose the following algorithm for finding a feasible $\delta$ that strives to maximize the minimum stride of any state. First, we use all the 1-stride tables as our initial selection. Second, for each $j$-var-stride ($j \geq 2$) table $t$ of state $s$, we create a tuple $(l, d, |t|)$ where $l$ denotes variable stride length, $d$ denotes the distance from state $s$ to the root of the deferment tree that $s$ belongs to, and $|t|$ denotes the number of entries in $t$. As stride length $l$ increases, the individual table size $|t|$ may increase significantly, particularly for the complete tables of root states. To balance table sizes, we set limits on the maximum allowed table size for root states and non-root states. If a root state table exceeds the root state threshold when we create its $j$-var-stride table, we apply self-loop unrolling once to its $(j-1)$-var-stride table to produce a $j$-var-stride table. If a non-root state table exceeds the non-root state threshold when we create its $j$-var-stride table, we simply use its $j-1$-var-stride table as its $j$-var-stride table. Third, we sort the tables by these tuple values in increasing order first using $l$, then using $d$, then using $|t|$, and finally a pseudorandom coin flip to break ties. Fourth, we consider each table $t$ in order. Let $t'$ be the table for the same state $s$ in the current selection. If replacing $t'$ by $t$ does not exceed our TCAM capacity $C$, we do the replacement.

## 6 Implementation and Modeling

| Entries | TCAM Chip size (36-bit wide) | TCAM Chip size (72-bit wide) | Latency ns |
|---|---|---|---|
| 1024 | 0.037 Mb | 0.074 Mb | 0.94 |
| 2048 | 0.074 Mb | 0.147 Mb | 1.10 |
| 4096 | 0.147 Mb | 0.295 Mb | 1.47 |
| 8192 | 0.295 Mb | 0.590 Mb | 1.84 |
| 16384 | 0.590 Mb | 1.18 Mb | 2.20 |
| 32768 | 1.18 Mb | 2.36 Mb | 2.57 |
| 65536 | 2.36 Mb | 4.72 Mb | 2.94 |
| 131072 | 4.72 Mb | 9.44 Mb | 3.37 |

Table 1: TCAM size in Mb and Latency in ns

We now describe some implementation issues associated with our TCAM based RE matching solution. First, the only hardware required to deploy our solution is the off-the-shelf TCAM (and its associated SRAM). Many deployed networking devices already have TCAMs, but these TCAMs are likely being used for other purposes. Thus, to deploy our solution on existing network devices, we would need to share an existing TCAM with another application. Alternatively, new networking devices can be designed with an additional dedicated TCAM chip.

Second, we describe how we update the TCAM when an RE set changes. First, we must compute a new DFA and its corresponding TCAM representation. For the moment, we recompute the TCAM representation from scratch, but we believe a better solution can be found and is something we plan to work on in the future. We report some timing results in our experimental section. Fortunately, this is an offline process during which time the DFA for the original RE set can still be used. The second step is loading the new TCAM entries into TCAM. If we have a second TCAM to support updates, this rewrite can occur while the first TCAM chip is still processing packet flows. If not, RE matching must halt while the new entries are loaded. This step can be performed very quickly, so the delay will be very short. In contrast, updating FPGA circuitry takes significantly longer.

We have not developed a full implementation of our system. Instead, we have only developed the algorithms that would take an RE set and construct the associated TCAM entries. Thus, we can only estimate the throughput of our system using TCAM models. We use Agrawal and Sherwood's TCAM model [1] assuming that each TCAM chip is manufactured with a $0.18\mu m$ process to compute the estimated latency of a single TCAM lookup based on the number of TCAM entries searched. These model latencies are shown in Table 1. We recognize that some processing must be done besides the TCAM lookup such as composing the next state ID with the next input character; however, because the TCAM lookup latency is much larger than any other operation, we focus only on this parameter when evaluating the potential throughput of our system.

## 7 Experimental Results

In this section, we evaluate our TCAM-based RE matching solution on real-world RE sets focusing on two metrics: TCAM space and RE matching throughput.

### 7.1 Methodology

We obtained 4 proprietary RE sets, namely C7, C8, C10, and C613, from a large networking vendor, and 4 public RE sets, namely Snort24, Snort31, Snort34, and Bro217 from the authors of [6] (we do report a slightly different number of states for Snort31, 20068 to 20052; this may be due to Becchi *et al.* making slight changes to their Regular Expression Processor that we used). Quoting Becchi *et al.* [6], "Snort rules have been filtered according to the headers ($HOME_NET, any, $EXTERNAL_NET, $HTTP_PORTS/any) and ($HOME_NET, any, 25, $HTTP_PORTS/any). In the experiments which follow, rules have been grouped so to obtain DFAs with reasonable size and, in parallel, have datasets with different characteristics in terms of number of wildcards, frequency of character ranges and so on." Of these 8 RE sets, the REs in C613 and Bro217 are all string matching REs, the REs in C7, C8, and C10 all contain wildcard closures '.*', and about 40% of the REs in Snort 24, Snort31, and Snort34 contain wildcard closures '.*'.

Finally, to test the scalability of our algorithms, we use one family of 34 REs from a recent public release of the Snort rules with headers ($EXTERNAL_NET, $HTTP_PORTS, $HOME_NET, any), most of which contain wildcard closures '.*'. We added REs one at a time until the number of DFA states reached 305,339. We name this family Scale.

We calculate TCAM space by multiplying the number of entries by the TCAM width: 36, 72, 144, 288, or 576 bits. For a given DFA, we compute a minimum width by summing the number of state ID bits required with the number of input bits required. In all cases, we needed at most 16 state ID bits. For 1-stride DFAs, we need exactly 8 input character bits, and for 7-var-stride DFAs, we need exactly 56 input character bits. We then calculate the TCAM width by rounding the minimum width up to the smallest larger legal TCAM width. For all our 1-stride DFAs, we use TCAM width 36. For all our 7-var-stride DFAs, we use TCAM width 72.

We estimate the potential throughput of our TCAM-based RE matching solution by using the model TCAM lookup speeds we computed in Section 6 to determine how many TCAM lookups can be performed in a second for a given number of TCAM entries and then multiplying this number by the number of characters processed per TCAM lookup. With 1-stride TCAMs, the number of characters processed per lookup is 1. For 7-var-stride DFAs, we measure the average number of characters processed per lookup in a variety of input streams. We use Becchi *et al.*'s network traffic generator [9] to generate a variety of synthetic input streams. This traffic generator includes a parameter that models the probability of malicious traffic $p_M$. With probability $p_M$, the next character is chosen so that it leads away from the start state. With probability $(1 - p_M)$, the next character is chosen uniformly at random.

| RE set | # states | TS | | | TS + TC2 | | | TS + TC4 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | TCAM megabits | #Entries per state | throughput Gbps | TCAM megabits | #Entries per state | thru Gbps | TCAM megabits | #Entries per state | thru Gbps |
| Bro217 | 6533 | 0.31 | 1.40 | 3.64 | 0.21 | 0.94 | 4.35 | 0.17 | 0.78 | 4.35 |
| C613 | 11308 | 0.63 | 1.61 | 3.11 | 0.52 | 1.35 | 3.64 | 0.45 | 1.17 | 3.64 |
| C10 | 14868 | 0.61 | 1.20 | 3.11 | 0.31 | 0.61 | 3.64 | 0.16 | 0.32 | 4.35 |
| C7 | 24750 | 1.00 | 1.18 | 3.11 | 0.53 | 0.62 | 3.64 | 0.29 | 0.34 | 3.64 |
| C8 | 3108 | 0.13 | 1.20 | 5.44 | 0.07 | 0.62 | 5.44 | 0.03 | 0.33 | 8.51 |
| Snort24 | 13886 | 0.55 | 1.16 | 3.64 | 0.30 | 0.64 | 3.64 | 0.18 | 0.38 | 4.35 |
| Snort31 | 20068 | 1.43 | 2.07 | 2.72 | 0.81 | 1.17 | 2.72 | 0.50 | 0.72 | 3.64 |
| Snort34 | 13825 | 0.56 | 1.18 | 3.11 | 0.30 | 0.62 | 3.64 | 0.17 | 0.36 | 4.35 |

Table 2: TCAM size and throughput for 1-stride DFAs

### 7.2 Results on 1-stride DFAs

Table 2 shows our experimental results on the 8 RE sets using 1-stride DFAs. We use TS to denote our transition sharing algorithm including both character bundling and shadow encoding. We use TC2 and TC4 to denote our table consolidation algorithm where we consolidate at most 2 and 4 transition tables together, respectively. For each RE set, we measure the number states in its 1-stride DFA, the resulting TCAM space in megabits, the average number of TCAM table entries per state, and the projected RE matching throughput; the number of TCAM entries is the number of states times the average number of entries per state. The TS column shows our results when we apply TS alone to each RE set. The TS+TC2 and TS+TC4 columns show our results when we apply both TS and TC under the consolidation limit of 2 and 4, respectively, to each RE set.

We draw the following conclusions from Table 2. (1) *Our RE matching solution is extremely effective in saving TCAM space*. Using TS+TC4, the maximum TCAM size for the 8 RE sets is only 0.50 Mb, which is two orders of magnitude smaller than the current largest commercially available TCAM chip size of 72 Mb. More specifically, the number of TCAM entries per DFA state ranges between .32 and 1.17 when we use TC4. We require 16, 32, or 64 SRAM bits per TCAM entry for TS, TS+TC2, and TS+TC4, respectively as we need to record 1, 2, or 4 state 16 bit state IDs in each decision, respectively. (2) *Transition sharing alone is very effective*. With the transition sharing algorithm alone, the maximum TCAM size is only 1.43Mb for the 8 RE sets. Furthermore, we see a relatively tight range of TCAM entries per state of 1.16 to 2.07. Transition sharing works extremely well with all 8 RE sets including those with wildcard closures and those with primarily strings. (3) *Table consolidation is very effective*. On the 8 RE sets, adding TC2 to TS improves compression by an average of 41% (ranging from 16% to 49%) where the maximum possible is 50%. We measure improvement by computing $(TS - (TS + TC2))/TS)$. Replacing TC2 with TC4 improves compression by an average of 36% (ranging from 13% to 47%) where we measure improvement by computing $((TS+TC2)-(TS+TC4))/(TS+TC2)$. Here we do observe a difference in performance, though. For the two RE sets Bro217 and C613 that are primarily strings without table consolidation, the average improvements of using TC2 and TC4 are only 24% and 15%, respectively. For the remaining six RE sets that have many wildcard closures, the average improvements are 47% and 43%, respectively. The reason, as we touched on in Section 4.4, is how wildcard closure creates multiple deferment trees with almost identical structure. Thus wildcard closures, the prime source of state explosion, is particularly amenable to compression by table consolidation. In such cases, doubling our table consolidation limit does not greatly increase SRAM cost. Specifically, while the number of SRAM bits per TCAM entry doubles as we double the consolidation limit, the number of TCAM entries required almost halves! (4) *Our RE matching solution achieves high throughput with even 1-stride DFAs*. For the TS+TC4 algorithm, on the 8 RE sets, the average throughput is 4.60Gbps (ranging from 3.64Gbps to 8.51Gbps).

We use our Scale dataset to assess the scalability of our algorithms' performance focusing on the number of TCAM entries per DFA state. Fig. 10(a) shows the number of TCAM entries per state for TS, TS+TC2, and TS+TC4 for the Scale REs containing 26 REs (with DFA size 1275) to 34 REs (with DFA size 305,339). The DFA size roughly doubled for every RE added. In general, the

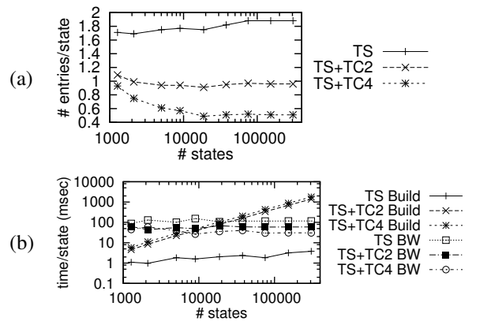



Figure 10: TCAM entries per DFA state (a) and compute time per DFA state (b) for Scale 26 through Scale 34.

number of TCAM entries per state is roughly constant and actually decreases with table consolidation. This is because table consolidation performs better as more REs with wildcard closures are added as there are more trees with similar structure in the deferment forest.

We now analyze running time. We ran our experiments on the Michigan State University High Performance Computing Center (HPCC). The HPCC has several clusters; most of our experiments were executed on the fastest cluster which has nodes that each have 2 quad-core Xeons running at 2.3GHz. The total RAM for each node is 8GB. Fig. 10(b) shows the compute time per state in milliseconds. The build times are the time per DFA state required to build the non-overlapping set of transitions (applying TS and TC); these increase linearly because these algorithms are quadratic in the number of DFA states. For our largest DFA Scale 34 with 305,339 states, the total time required for TS, TS+TC2, and TS+TC4 is 19.25 mins, 118.6 hrs, and 150.2 hrs, respectively. These times are cumulative; that is going from TS+TC2 to TS+TC4 requires an additional 31.6 hours. This table consolidation time is roughly one fourth of the first table consolidation time because the number of DFA states has been cut in half by the first table consolidation and table consolidation has a quadratic running time in the number of DFA states. The BW times are the time per DFA state required to minimize these transition tables using the Bitweaving algorithm in [21]; these times are roughly constant as Bitweaving depends on the size of the transition tables for each state and is not dependent on the size of the DFA. For our largest DFA Scale 34 with 305,339 states, the total Bitweaving optimization time on TS, TS+TC2, and TS+TC4 is 10 hrs, 5 hrs, and 2.5 hrs. These times are not cumulative and fall by a factor of 2 as each table consolidation step cuts the number of DFA states by a factor of 2.

### 7.3 Results on 7-var-stride DFAs

We consider two implementations of variable striding assuming we have a 2.36 megabit TCAM with TCAM width 72 bits (32,768 entries). Using Table 1, the latency of a lookup is 2.57 ns. Thus, the potential RE matching throughput of by a 7-var-stride DFA with average stride $S$ is $8 \times S/.00000000257 = 3.11 \times S$ Gbps.

In our first implementation, we only use self-loop unrolling of root states in the deferment forest. Specifically, for each RE set, we first construct the 1-stride DFA using transition sharing. We then apply self-loop unrolling to each root state of the deferment forest to create a 7-var-stride transition table. In all cases, the increase in size due to self-loop unrolling is tiny. The bigger issue was that the TCAM width doubled from 36 bits to 72 bits. We can decrease the TCAM space by using table consolidation; this was very effective for all RE sets except the string matching RE sets Bro217 and C613. This was only necessary for Snort31. All other self-loop unrolled tables fit within our available TCAM space.

Second, we apply full variable striding. Specifically, we first create 1-stride DFAs using transition sharing and then apply variable striding with no table consolidation, table consolidation with 2-decision tables, and table consolidation with 4-decision tables. We use the best result that fits within the 2.36 megabit TCAM space. For the RE sets Bro217, C8, C613, Snort24 and Snort34, no table consolidation is used. For C10 and Snort31, we use table consolidation with 2-decision tables. For C7, we use table consolidation with 4-decision tables.

We now run both implementations of our 7-var-stride DFAs on traces of length 287484 to compute the average stride. For each RE set, we generate 4 traces using Becchi *et al.*'s trace generator tool using default values 35%, 55%, 75%, and 95% for the parameter $p_M$. These generate increasingly malicious traffic that is more likely to move away from the start state towards distant accept states of that DFA. We also generate a completely random string to model completely uniform traffic such as binary traffic patterns which we treat as $p_M = 0$.

We group the 8 RE sets into 3 groups: group (a) represents the two string matching RE sets Bro217 and C613; group (b) represents the three RE sets C7, C8, and C10 that contain all wildcard closures; group (c) represents the three RE sets Snort24, Snort31, and Snort34 that contain roughly 40% wildcard closures. Fig. 11 shows the average stride length and throughput for the three groups of RE sets according to the parameter $p_M$ (the random string trace is $p_M = 0$).

We make the following observations. *(1) Self-loop unrolling is extremely effective on the uniform trace.* For the non string matching sets, it achieves an average stride length of 5.97 and 5.84 and RE matching throughputs



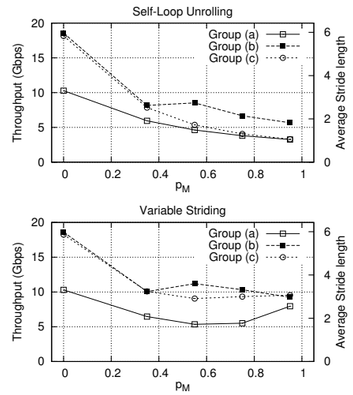



Figure 11: The throughput and average stride length of RE sets.

of 18.58 and 18.15 Gbps for groups (b) and (c), respectively. For the string matching sets in group (a), it achieves an average stride length of 3.30 and a resulting throughput of 10.29 Gbps. Even though only the root states are unrolled, self-loop unrolling works very well because the non-root states that defer most transitions to a root state will still benefit from that root state's unrolled self-loops. In particular, it is likely that there will be long stretches of the input stream that repeatedly return to a root state and take full advantage of the unrolled self-loops. *(2) The performance of self-loop unrolling does degrade steadily as $p_M$ increases for all RE sets except those in group (b).* This occurs because as $p_M$ increases, we are more likely to move away from any default root state. Thus, fewer transitions will be able to leverage the unrolled self-loops at root states. *(3) For the uniform trace, full variable striding does little to increase RE matching throughput.* Of course, for the non-string matching RE sets, there was little room for improvement. *(4) As $p_M$ increases, full variable striding does significantly increase throughput, particularly for groups (b) and (c).* For example, for groups (b) and (c), the minimum average stride length is 2.91 for all values of $p_M$ which leads to a minimum throughput of 9.06Gbps. Also, for all groups of RE sets, the average stride length for full variable striding is much higher than that for self-loop unrolling for large $p_M$. For example, when $p_M = 95\%$, full variable striding achieves average stride lengths of 2.55, 2.97, and 3.07 for groups (a), (b), and (c), respectively, whereas self-loop unrolling achieves average stride lengths of only 1.04, 1.83, and 1.06 for groups (a), (b), and (c), respectively.

These results indicate the following. First, self-loop unrolling is extremely effective at increasing throughput for random traffic traces. Second, other variable striding techniques can mitigate many of the effects of malicious traffic that lead away from the start state.

## 8 Conclusions

We make four key contributions in this paper. (1) We propose the first TCAM-based RE matching solution. We prove that this unexplored direction not only works but also works well. (2) We propose two fundamental techniques, transition sharing and table consolidation, to minimize TCAM space. (3) We propose variable striding to speed up RE matching while carefully controlling the corresponding increase in memory. (4) We implemented our techniques and conducted experiments on real-world RE sets. We show that small TCAMs are capable of storing large DFAs. For example, in our experiments, we were able to store a DFA with 25K states in a 0.5Mb TCAM chip; most DFAs require at most 1 TCAM entry per DFA state. With variable striding, we show that a throughput of up to 18.6 Gbps is possible.

## References

[1] B. Agrawal and T. Sherwood. Modeling TCAM power for next generation network devices. In *Proc. IEEE Int. Symposium on Performance Analysis of Systems and Software*, 2006.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 1975.

[3] M. Alicherry, M. Muthuprasanna, and V. Kumar. High speed pattern matching for network IDS/IPS. In *Proc. ICNP*, 2006.

[4] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *Proc. INFOCOM*, 2007.

[5] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proc. CoNext*, 2007.

[6] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proc. ANCS*, 2007.

[7] M. Becchi and P. Crowley. Efficient regular expression evaluation: Theory to practice. In *Proc. ANCS*, 2008.

[8] M. Becchi and P. Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *Proc. CoNEXT*, 2008.

[9] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *Proc. IEEE IISWC*, 2008.

[10] M. Becchi, C. Wiseman, and P. Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *Proc. ANCS*, 2009.

[11] A. Bremler-Bar, D. Hay, and Y. Koral. CompactDFA: generic state machine compression for scalable pattern matching In *Proc. INFOCOM*, 2010.

[12] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *SIGARCH Computer Architecture News*, 2006.

[13] C. R. Clark and D. E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *Proc. FPL*, pages 956–959, 2003.

[14] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In *FCCM* 2004.

[15] J. E. Hopcroft. *The Theory of Machines and Computations*, chapter An nlogn algorithm for minimizing the states in a finite automaton, pages 189–196. Academic Press, 1971.

[16] S. Kong, R. Smith, and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *Proc. ACM SecureComm*, Article 1, 2008.

[17] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proc. ACM/IEEE ANCS*, pages 155–164, 2007.

[18] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. SIGCOMM*, 2006.

[19] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proc. ANCS*, pages 81–92, 2006.

[20] C. R. Meiners, A. X. Liu, and E. Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. In *Proc. ICNP*, 2007.

[21] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. In *Proc. ICNP*, 2009.

[22] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proc. ACM/IEEE ANCS*, 2007.

[23] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an internet firewall. In *FCCM*, 2003.

[24] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In *FCCM*, 2001.

[25] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *Proc. Symposium on Security and Privacy*, 2008.

[26] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proc. SIGCOMM*, pages 207–218, 2008.

[27] R. Sommer and V. Paxson. Enhancing bytelevel network intrusion detection signatures with context. In *Proc. ACM CCS*, pages 262–271, 2003.

[28] I. Sourdis and D. Pnevmatikatos. Pnevmatikatos: Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In *Proc. FCCM*, pages 880–889, 2003.

[29] I. Sourdis and D. Pnevmatikatos. Pre-decoded cams for efficient and high-speed nids pattern matching. In *Proc. FCCM*, 2004.

[30] J.-S. Sung, S.-M. Kang, Y. Lee, T.-G. Kwon, and B.-T. Kim. A multi-gigabit rate deep packet inspection algorithm using TCAM. In *Proc. IEEE GLOBECOM*, 2005.

[31] S. Suri, T. Sandholm, and P. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 2003.

[32] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proc. ISCA*, 2005.

[33] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proc. IEEE Infocom*, pages 333–340, 2004.

[34] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proc. ANCS*, 2006.

[35] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *Proc. ICNP*, 2004.



# Searching the Searchers with SearchAudit

*John P. John*[‡§], *Fang Yu*[§], *Yinglian Xie*[§], *Martín Abadi*[§*], *Arvind Krishnamurthy*[‡]
[‡]*University of Washington* [§]*Microsoft Research Silicon Valley*
{*jjohn, arvind*}*@cs.washington.edu* {*fangyu, yxie, abadi*}*@microsoft.com*
[*]*University of California, Santa Cruz*

## Abstract

Search engines not only assist normal users, but also provide information that hackers and other malicious entities can exploit in their nefarious activities. With carefully crafted search queries, attackers can gather information such as email addresses and misconfigured or even vulnerable servers.

We present SearchAudit, a framework that identifies malicious queries from massive search engine logs in order to uncover their relationship with potential attacks. SearchAudit takes in a small set of malicious queries as seed, expands the set using search logs, and generates regular expressions for detecting new malicious queries. For instance, we show that, relying on just 500 malicious queries as seed, SearchAudit discovers an additional 4 million distinct malicious queries and thousands of vulnerable Web sites. In addition, SearchAudit reveals a series of phishing attacks from more than 400 phishing domains that compromised a large number of Windows Live Messenger user credentials. Thus, we believe that SearchAudit can serve as a useful tool for identifying and preventing a wide class of attacks in their early phases.

## 1 Introduction

With the amount of information in the Web rapidly growing, the search engine has become an everyday tool for people to find relevant and useful information. While search engines make online browsing easier for normal users, they have also been exploited by malicious entities to facilitate their various attacks. For example, in 2004, the *MyDoom* worm used Google to search for email addresses in order to send spam and virus emails. Recently, it was also reported that hackers used search engines to identify vulnerable Web sites and compromised them immediately after the malicious searches [20, 16]. These compromised Web sites were then used to serve malware or phishing pages.

Indeed, by crafting specific search queries, hackers may get very specific information from search engines that could potentially reveal the existence and locations of security flaws such as misconfigured servers and vulnerable software. Furthermore, attackers may prefer using search engines because it is stealthier and easier than setting up their own crawlers.

The identification of these malicious queries thus provides a wide range of opportunities to disrupt or prevent potential attacks at their early stages. For example, a search engine may choose not to return results to these malicious queries [20], making it harder for attackers to obtain useful information. In addition, these malicious queries could provide rich information about the attackers, including their intentions and locations. Therefore, strategically, we can let the attackers guide us to better understand their methods and techniques, and ultimately, to predict and prevent followup attacks before they are launched.

In this paper, we present *SearchAudit*, a suspicious-query generation framework that identifies malicious queries by auditing search engine logs. While auditing is often an important component of system security, the auditing of search logs is particularly worthwhile, both because authentication and authorization (two other pillars of security [14]) are relatively weak in search engines, and because of the wealth of information that search engines and their logs contain.

Working with SearchAudit consists of two stages: identification and investigation. In the first stage, SearchAudit identifies malicious queries. In the second stage, with SearchAudit's assistance, we focus on analyzing those queries and the attacks of which they are part.

More specifically, in the first stage, SearchAudit takes a few known malicious queries as seed input and tries to identify more malicious queries. The seed can be obtained from hacker Web sites [1], known security vulnerabilities, or case studies performed by other security

researchers [16]. As seed malicious queries are usually limited in quantity and restricted by previous discoveries, SearchAudit monitors the hosts that conducted these malicious queries to obtain an expanded set of queries from these hosts. Using the expanded set of queries, SearchAudit further generates regular expressions, which are then used to match search logs for identifying other malicious queries. This step is critical as malicious queries are typically automated searches generated by scripts. Using regular expressions offers us the opportunity to catch a large number of other queries with a similar format, possibly generated by such scripts.

After identifying a large number of malicious queries, in stage two, we analyze the malicious queries and the correlation between search and other attacks. In particular, we ask questions such as: why do attackers use Web search, how do they leverage search results, and who are the victims. Answers to these questions not only help us better understand the attacks, but also provide us an opportunity to protect or notify potential victims before the actual attacks are launched, and hence stop attacks in their early stages.

We apply SearchAudit to three months of sampled Bing search logs. As search logs contain massive amounts of data, SearchAudit is implemented on the Dryad/DryadLINQ [11, 26] platform for large-scale data analysis. It is able to process over 1.2TB of data in 7 hours using 240 machines.

To our knowledge, we are the first to present a systematic approach for uncovering the correlations between malicious searches and the attacks enabled by them. Our main results include:

- **Enhanced detection capability:** Using just 500 seed queries obtained from one hacker Web site, SearchAudit detects another 4 million malicious queries, some even before they are listed by hacker Web sites.
- **Low false-positive rates**. Over 99% of the captured malicious queries display multiple bot features, while less than 2% of normal user queries do.
- **Ability to detect new attacks:** While the seed queries are mostly ones used to search for Web site vulnerabilities, SearchAudit identifies a large number of queries belonging to a different type of attack—forum spamming.
- **Facilitation of attack analysis:** SearchAudit helps identify vulnerable Web sites that are targeted by attackers. In addition, SearchAudit helps analyze a series of phishing attacks that lasted for more than one year. These attacks set up more than 400 phishing domains, and tried to steal a large number of Windows Live Messenger user credentials.

The rest of the paper is organized as follows. We start with reviewing related work in Section 2. Then we present the architecture of SearchAudit in Section 3. As SearchAudit contains two stages, Section 4 focuses on the results of the first stage—presenting the malicious queries identified, and verifying that they are indeed malicious. Section 5 describes the second stage of SearchAudit—analyzing the correlation between malicious queries and other attacks. In this paper, we study three types of attacks in detail: searching for vulnerable Web sites (Section 6), forum spamming (Section 7), and Windows Live Messenger phishing attacks (Section 8). Finally we conclude in Section 9.

## 2 Related Work

There is a significant amount of automated Web traffic on the Internet [5]. A recent study by Yu et al. showed that more than 3% of the entire search traffic may be generated by stealthy search bots [25] .

One natural question to ask is: what is the motivation of these search bots? While some search bots have legitimate uses, e.g., by search engine competitors or third parties for studying search quality [8, 17], many others could be malicious. It is widely known that attackers conduct click fraud for monetary gain [7, 10]. Recently, researchers have associated malicious searches with other types of attacks. For example, Provos et al. reported that worms such as MyDoom.O and Santy used Web search to identify victims for spreading infection [20]. Also, Moore et al. [16] identified four types of evil searches and showed that some Web sites were compromised shortly after evil searches. They showed that attackers searched for keywords like *"phpizabi v0.848b c1 hfp1"* to gather all the Web sites that have a known PHP vulnerability [9]. Subsequently these vulnerable Web servers were compromised to set up phishing pages.

Besides email spamming and phishing, there are many other types of attacks, e.g., malware propagation and Denial of Service (DoS) attacks. Although there are a wealth of attack-detection approaches, most of these attacks were studied in isolation. Their correlations, especially to Web searches, have not been extensively studied. In this paper, we aim to take a step towards a systematic framework to unveil the correlations between malicious searches and many other attacks.

In SearchAudit, we derive regular expression patterns for matching malicious queries. There are many existing signature-generation techniques for detecting worms and spam emails such as Polygraph [18], Hamsa [15], Autograph [12], Earlybird [21], Honeycomb [13], Neman [24] Vigilante [6], and AutoRE [23]. Some of these approaches are based on semantics, e.g., Neman and Vigilante, and hence are not suitable for us, since query strings do not have semantic information. The remaining content-based signature-generation schemes, Honeycomb, Polygraph, Hamsa, and AutoRE, can generate string tokens or regular expressions. These are more appealing to us since attackers add random keywords to query strings, and we want the generated signatures to capture this polymorphism. In this work, we choose AutoRE, which generates regular expression signatures.

In [20], Provos et al. found malicious queries from the Santy worm by looking at search results. In those attacks, the attackers constantly changed the queries, but obtained similar search results (viz., the Web servers that are vulnerable to Santy's attack). SearchAudit, on the other hand, is primarily targeted at finding new attacks, of which we have no prior knowledge. SearchAudit is thus a general framework to detect and understand malicious searches. While there might already be proprietary approaches adopted by various search engines, or anecdotal evidence of malicious searches, we hope that our analysis results can provide useful information to the general research community.

## 3 Architecture

Our main goal is to let attackers be our guides—to follow their activities and predict their future attacks. We use a small-sized set of seed activities to bootstrap our system. The seed is usually limited and restricted to malicious searches of which we are aware. The system then applies a sequence of techniques to extend this seed set in order to identify previously unknown attacks and obtain a more comprehensive view of malicious search behavior.

Figure 1 presents the architecture of our system. At a high level, the system can be viewed as having two stages. In the first stage, it examines search query logs, and expands the set of seed queries to generate additional sets of suspicious queries. This stage is automated and quite general, i.e., it can be used to find different types of suspicious queries pertaining to different malicious activities. The second stage involves the analysis of these suspicious queries to see how different attacks are connected with search—this is mostly done manually, since it requires a significant amount of domain knowledge to understand the behavior of the different malicious entities. This section focuses on the first stage of our system and Sections 6, 7, and 8 provide examples of the analysis done in the second stage.

Extending the seed using query logs appears to be a straightforward idea. Yet, there are two challenges. First, hackers do not always use the same queries; they modify and change query terms over time in order to obtain different sets of search results, and thereby identify new victims. Therefore, simply using a blacklist of bad queries is not effective. Second, malicious searches may be mixed with normal user activities, especially on proxies. So we need to differentiate malicious queries from normal ones, though they may originate from the same machine or IP address. To address these challenges, we do not simply use the suspicious queries directly, but instead generate regular expression signatures from these suspicious queries. Regular expressions help us capture the structure of these malicious queries, which is necessary to identify future queries. We also filter regular expressions that are too general and therefore match both malicious and normal queries. Using these two approaches, the first stage of the system now consists of a pipeline of two steps: *Query Expansion* and *Regular Expression Generation*. Since any set of malicious queries could potentially lead to additional ones, we loop back these queries until we reach a fixed point with respect to query expansion. The rest of this section presents each of the stages in detail.

### 3.1 Query Expansion

The first step in our system is to take a small set of seed queries and expand them. These seed queries are known to be suspicious or malicious. They could be obtained from a variety of sources, such as preliminary analysis of the search query logs or with the help of domain experts.

Our search logs contain the following information: a query, the time at which the query was issued, the set of results returned to the searcher, and a few properties of the request, such as the IP address that issued the request and the user agent (which identifies the Web browser used). Since the amount of data in the search logs is massive, we use the Dryad/DryadLINQ platform to process data in parallel on a cluster of hundreds of machines.

The seed queries are expanded as follows. We run the seed queries through the search logs to find exact query matches. For each record where the queries match exactly, we extract the IP address that issued the query. We then go back to the search logs and extract all queries that were issued by this IP address. The reasoning here is that since this IP address issued a query that we believe to be malicious, it is probably that other queries from this IP address would also be malicious. This is because attackers typically issue not just a single query but rather multiple queries so as to get more search results. This method of expansion would allow us to capture the other queries issued.

However, it must be noted that since we are using the IP address to expand to other queries, we need to be careful about dynamic IP addresses because of DHCP. In order to reduce the impact of dynamic IPs on our data, we consider only queries that were made on the same day as the seed query.

At the end of this step, we have all the queries that were issued from suspicious IP addresses on the same day.

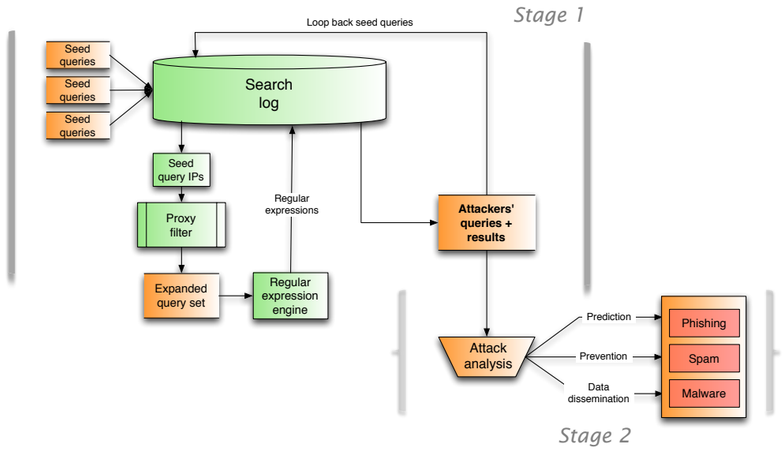



Figure 1: The architecture of the system is a pipeline connecting the query expansion framework, the proxy elimination, and the regular expression generation.

## 3.2 Regular Expression Generation

The next step after performing query expansion is the generation of regular expressions. We prefer regular expressions over fixed strings for two reasons. First, they can potentially match malicious searches even if attackers change the search terms slightly. In our logs, we find that many hackers add restrictions to the query terms, e.g., adding "`site:cn`" will obtain search results in the `.cn` domain only; regular expressions can capture these variations of queries. Second, as many of the queries are generated using scripts, regular expressions can capture the structure of the queries and therefore can match future malicious queries.

**Signature Generation:** We use a technique similar to AutoRE [23] to derive regular expressions, with a few modifications to incorporate additional information from the search domain, such as giving importance to word boundaries and special characters in a query. The regular-expression generator works as follows. First, it builds a suffix array to identify all popular keywords in the input set. Then it picks the most popular keyword and builds a root node that contains all the input strings matching this keyword. For the remaining strings, it repeats the process of selecting root nodes until all strings are selected. These root nodes are used to start building trees of frequent substrings. Then the regular-expression generator recursively processes each tree to form a forest. For each tree node, the keywords on the path to the root construct a pattern. It then checks the content between keywords and places restrictions on it (e.g., [0-9]{1,3} to constrain the intervening content to be one to three digits). In addition, for each regular expression, we compute a score that measures the likelihood that the regular expression would match a random string. This score is based on entropy analysis, as described in [23]; the lower the score, the more specific the regular expression. However, a too specific regular expression would be equivalent to having an exact match, and thus loses the benefit of using the regular expression in the first place. We therefore need a score threshold to pick the set of regular expressions in order to trade off between the specificity of the regular expression and the possibility of it matching too many benign queries. In SearchAudit, we select regular expressions with score lower than 0.6. (Parameter selection is discussed in detail in Section 4.2.)

**Eliminating Redundancies:** One issue with the generated regular expressions is that some of them may be redundant, i.e., though not identical, they match the same or similar set of queries. For example, three input strings `query site:A`, `query site:B`, and `query` may generate two regular expressions `query.{0,7}` and `query site:.{1}`. The two regular expressions have different coverage and scores, but are both valid. In order to eliminate redundancy in regular expressions, we use the REGEX_CONSOLIDATE algorithm described in Algorithm 1. The algorithm takes as input $S$, the set of input queries, $R_1, \ldots, R_n$, the regular expressions, and returns $R$, the subset of input regular expressions. Here, the function MATCHES($S$, $R_i$) returns the strings $V \subseteq S$ that match the regular expression $R_i$.

We note that REGEX_CONSOLIDATE is a greedy algorithm and does not return the minimal set of regular expressions required to match all the input strings. Finding the minimal set is in fact NP-Hard [4].

**Algorithm 1** REGEX_CONSOLIDATE($S, R_1, \ldots, R_n$)

$R \leftarrow \{\}$
$V \leftarrow \cup_{i=1}^{n}$ MATCHES($S$, $R_i$)
**while** $|V| > 0$ **do**
  $R_{max} \leftarrow R_j$ where $R_j$ is the regular expression that matches the most number of strings in $V$
  $R \leftarrow R \cup R_{max}$
  $V \leftarrow V -$ MATCHES($V$, $R_{max}$)
**end while**
**return** $R$

This ability to consolidate regular expressions has another advantage: if the input to the regular-expression generator contains too many strings, it is split into multiple groups, and regular expressions are generated for each group separately. These regular expressions can then be merged together using REGEX_CONSOLIDATE.

**Eliminating Proxies:** We observe that we can speed up the generation of regular expressions by reducing the number of strings fed as input to the regular-expression generator. However, we would like to do this without sacrificing the quality of the regular expressions generated. We observe in our experiments that some of the seed malicious queries are performed by IP addresses that correspond to public proxies or NATs. These IPs are characterized by a large query volume, since the same IP is used by multiple people. Also, most of the queries from these IPs are regular benign queries, interspersed with a few malicious ones. Therefore, eliminating these IPs would provide a quick and easy way of decreasing the number of input strings, while still leaving most of the malicious queries untouched.

In order to detect such *proxy-like* IPs, we use a simple heuristic called *behavioral profiling*. Most users in a geographical region have similar query patterns, which are different from that of an attacker. For proxies that have mostly legitimate users, their set of queries will have a large overlap with the popular queries from the same `/16` IP prefix. We label an IP as a proxy if it issues more than 1000 queries in a day, and if the $k$ most popular queries from that IP and the $k$ most popular queries from that prefix overlap in $m$ queries. (We empirically find $k = 100$ and $m = 5$ to work well.) Note however, that the proxy elimination is purely a performance optimization, and not necessary for the correct operation of SearchAudit. Behavioral profiling could also be replaced with a better technique for detecting legitimate proxies.

**Looping Back Queries:** Once the regular expressions are generated, they are applied to the search logs in order to extract all queries that match the regular expressions. This is an enlarged set of suspicious queries. These queries generated by SearchAudit can now be fed back into the system as new seed queries for another iteration. A discussion on the effect of looping back queries as seeds, and its benefits, is presented in Section 4.3.3.

| Matching Type | Total Queries | Uniq. Queries | IPs |
|---|---|---|---|
| Seed match | 122,529 | 122 | 174 |
| Exact match (expanded) | 216,000 | 800 | 264 |
| Regular expression match | 297,181 | 3,560 | 1,001 |

Table 1: The number of search requests, unique queries, and IPs for different matching techniques on the February 2009 dataset.

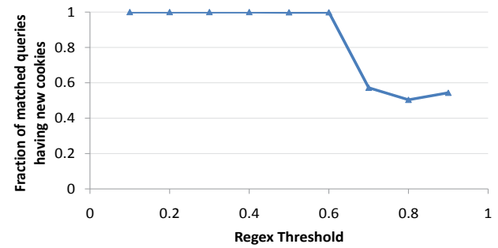



Figure 2: Selecting the threshold for regular expression scores: for regular expressions having score 0.6 or less, nearly all the matched queries have new cookies.

## 4 Stage One Results

We apply SearchAudit to several months of search logs in order to identify malicious searchers. In this section, we first describe the data collection and system setup. Then we explain the process of parameter selection. Finally, we present the detection results and verify the results.

### 4.1 Data Description and System Setup

We use three months of search logs from the Bing search engine for our study: February 2009 (when it was known as Live Search), December 2009, and January 2010. Each month of sampled data contains around 2 billion pageviews. Each pageview records all the activities related to a search result page, including information such as the query terms, the links clicked, the query IP address, the cookie, the user agent, and the referral URL. Because of privacy concerns, the cookie and the user agent fields are anonymized by hashing.

The seed malicious queries are obtained from a hacker Web site `milw0rm.com` [1]. We crawl the site and extract 500 malicious queries, which were posted between May 2006 and August 2009.

We implement SearchAudit on the Dryad/DryadLINQ platform, where data is processed in parallel on a cluster of 240 machines. The entire process of SearchAudit takes about 7 hours to process the 1.2 TB of sampled data.

## 4.2 Selection of Regular Expressions

As described in Section 3.2, we can eliminate proxies to speed up the regular expression generation. If we do not eliminate proxies, the input to the regular-expression generator can contain queries from the proxies, and there may be many benign queries among them. As a result, although some of the generated regular expressions may be specific, they could match benign queries. In this setting, we need to examine each regular expression individually, and select those that match only malicious queries. To do this, we use the presence of old cookies to guide us. We observe that if we pick a random set of search queries (which may contain a mix of normal and malicious queries), the number of new cookies in them is substantially low. However, for the known malicious queries (the seed queries), it is close to 100%, because most automated traffic either does not enable cookies or presents invalid cookies. (In both these cases, a new cookie is created by the search engine and assigned to the search request.) Of course, cookie presence is just one feature of regular user queries. We can use other features as well, as discussed in Section 4.5.

If proxies are eliminated, the remaining queries are from the attackers' IPs, and we find that most of them are malicious. In this case, we can simply use a threshold to pick regular expressions based on their scores. This threshold represents a trade-off between the specificity of the regular expression and the possibility of it being too general and matching too many random queries. Again, we use the number of new cookies as a metric to guide us in our threshold selection. Figure 2 shows the relationship between the regular expression score and the percentage of new cookies in the queries matched by the regular expressions. We see empirically that expressions with scores lower than 0.6 have a very high fraction of new cookies ($> 99.85\%$), similar to what we observe with the seed malicious queries. On the other hand, regular expressions with score greater than 0.6 match queries where the fraction of new cookies is similar to what we see for a random sampling of user queries; therefore it is plausible that these regular expressions mostly match random queries that are not necessarily malicious.

In our tests, proxy elimination filters most of the benign queries, but less than 3% of the unique malicious queries (using cookie-age as the indicator). Therefore it has little effect on the generated regular expressions. Consequently, all the results presented in the paper are with the use of proxy elimination. We choose 0.6 as the regular expression threshold, and this ends up picking about 20% of the generated regular expressions.

| Seed Queries Used | Coverage |
|---|---|
| 100 queries (pre-2009) | 100% |
| Random 50% | 98.50% |
| Random 25% | 88.50% |

Table 2: Malicious query coverage obtained when using different subsets of the seed queries.

## 4.3 Detection Results

We now present results obtained from running SearchAudit, and show how each component contributes to the end results.

### 4.3.1 Effect of Query Expansion and Regular Expression Matching

We feed the 500 malicious queries obtained from `milw0rm.com` into SearchAudit, and examine the February 2009 dataset. Using exact string match, we find that 122 of the 500 queries appear in the dataset, and we identify 174 IP addresses that issued these queries. Many of these queries are submitted from multiple IP addresses and many times, presumably to fetch multiple pages of search results. In all, there are 122,529 such queries issued by these IP addresses to the search engine. Then we use the query expansion module together with the proxy elimination module of SearchAudit and obtain 800 unique queries from 264 IP addresses. Finally we run these queries through the regular expression generation engine.

Table 1 quantifies the number of additional queries SearchAudit identifies by the use of query expansion and regular expression generation. Using regular expression matching, SearchAudit identifies 3,560 distinct malicious queries from 1001 IP addresses. Compared to exact matching of the seed queries, regular-expression-based matching increases the number of unique queries found by almost a factor of 30. We also find 4 times more attacker IPs. Thus using regular expressions for matching provides significant gains.

### 4.3.2 Effect of Incomplete Seeds

Seed queries are inherently incomplete, since they are a very small set of known malicious queries. In this section, we look at how much coverage SearchAudit continues to get when the number of seed queries is decreased.

First, we split the 122 seed queries into two sets: 100 queries that were first posted on `milw0rm.com` before 2009, and the remaining 22 that were posted in 2009. We then use the 100 queries as our seed, and run SearchAudit on the same search log for a week in February 2009. We find that the queries generated by SearchAudit recover all the 122 seed queries. Therefore SearchAudit is effective in finding the malicious queries even before they are posted on the Web site; in fact we find queries in the search logs several months before they are first posted on the Web site.

Next, we choose a random subset of the original seed queries. With 50% of the randomly selected seed queries, our coverage is 98.5% out of the 122 input seed queries; and using just 25% of the seed queries, we can obtain 88.5% of the queries. These results are summarized in Table 2.

| | IPs | Queries | % Queries with Cookies |
|---|---|---|---|
| No loopback | 1,001 | 297,181 | 0.15% |
| Loopback 1 | 39,969 | 8,992,839 | 0.87% |
| Loopback 2 | 40,318 | 9,001,737 | 0.96% |
| Loopback 3 | 41,301 | 9,028,143 | 0.97% |

Table 3: The number of IPs and queries captured by SearchAudit in the February 2009 dataset, with and without looping back.

### 4.3.3 Looping Back Seed Queries

After SearchAudit is bootstrapped using malicious queries, it uses the derived regular expressions to generate a steady stream of queries that are being performed by attackers. SearchAudit uses these as new seeds to generate additional suspicious queries. Each such set of suspicious queries can subsequently be fed back as new seed input to SearchAudit, until the system reaches a fixed point, or until the marginal benefit of finding more such queries outweighs the cost.

To measure when this fixed point would occur, we use the February 2009 dataset, and run SearchAudit multiple times, each time taking the output from the previous run as the seed input. For the first run, we use the 500 seed queries obtained from `milw0rm.com`.

Table 3 summarizes our findings. We see that, as expected, the number of queries captured increases when the generated queries are looped back as new seeds. Also, the number of queries that have valid cookies remains quite small throughout ($< 1\%$), suggesting that the new queries generated through the loopback are similar to the seed queries and the queries generated in the first round. We observe that looping back once significantly increases the set of queries and IPs captured (from 1001 IPs to almost 40,000 IPs), but subsequent iterations do not add much information.

Therefore, we restrict SearchAudit to loop back the generated queries as seeds exactly once.

| Dataset | IPs | Total Queries | Uniq. Queries |
|---|---|---|---|
| Feb-2009 | 39,969 | 8,992,839 | 542,505 |
| Dec-2009 | 29,364 | 5,824,212 | 3,955,244 |
| Jan-2010 | 42,833 | 2,846,703 | 422,301 |

Table 4: The number of search requests, unique queries, and IPs captured by SearchAudit in the different datasets.

### 4.3.4 Overall Matching Statistics

Putting it all together, i.e., using regular expression matching and loopback, Table 4 shows the number of IPs, total queries, and distinct queries that SearchAudit identifies in each of the datasets. Overall, SearchAudit identifies over 40,000 IPs issuing more than 4 million malicious queries, resulting in over 17 million pageviews. One interesting point to note here is the significant spike in the number of unique queries found in the December dataset. The reason for this spike is the presence of a set of attacker IPs that do not fetch multiple result pages for a query, but instead generate new queries by adding a random dictionary word to the query, thereby increasing the number of distinct queries we observe.

## 4.4 Verification of Malicious Queries

Next, we verify that the queries identified by SearchAudit are indeed malicious queries. As we lack ground truth information about whether a query is malicious or not, we adopt two approaches. The first is to check whether the query is reported on any hacker Web sites or security bulletins. The second is to check query behavior—whether the query matches individual bot or botnet features.

For each query $q$ returned by SearchAudit, we issue a query "$q$ AND (dork OR vulnerability)" to the search engine, and save the results. Here, the term "dork" is used by attackers to represent malicious searches. We add the terms "dork" and "vulnerability" to the query to help us find forums and Web sites that discuss these queries. We then look at the most popular domains appearing in the search results across multiple queries. Domains that list a large number of malicious searches from our set are likely to be security forums, blogs by security companies or researchers, or even hacker Web sites. These can now be used as new sources for finding more seed queries. We manually examine 50 of these Web sites, and find that around 60% of them are security blogs or advisories. The remaining 40% are in fact hacker forums. In all, 73% of the queries reported by SearchAudit contain search results associated with these 50 Web sites.

Next we look at two sets of behavioral features that would indicate whether the query is automated, and whether a set of queries was generated by the same

script. The first set of features applies to individual bot-generated queries, e.g., not clicking any link. They indicate whether a query is likely to be scripted or not. The second set of features relates to botnet group properties. In particular, they quantify the likelihood that the different queries captured by a particular regular expression were generated by the same (or similar) script.

Note that although these behavior features could distinguish bot queries from human-generated ones, they are not robust features because attackers can easily use randomization or change their behavior if they know these features. In this work, we use these behavior features only for validation rather than relying on them to detect malicious queries.

#### 4.4.1 Verification of Queries Generated by Individual Bots

To distinguish bot queries from those generated by human users, we select the following features:

- *Cookie:* This is the cookie presented in the search request. Most bot queries do not enable cookies, resulting in an empty cookie field. For normal users who do not clear their cookies, all the queries carry the old cookies.
- *Link clicked:* This records whether any link in the search results was clicked by the user. Many bots do not click any link on the result page. Instead, they scrape the results off the page.

We compare queries returned by SearchAudit with queries issued by normal users for popular terms such as `facebook` and `craigslist`. Table 5 and Table 6 show the comparison results. We see that for SearchAudit returned queries, 98.8% of them disable cookies, as opposed to normal users, where only 2.7% disable cookies. We also see that on average, all the queries in a group returned by SearchAudit had no links clicked. On the other hand, for normal users, over 85% of the searches resulted in clicks. All these common features suggest that the queries returned by SearchAudit are highly likely to be automated or scripted searches, rather than being submitted by regular users.

#### 4.4.2 Verification of Queries Generated by Botnets

Having shown that individual queries identified by SearchAudit display bot characteristics, we next study whether a set of queries matched by a regular expression are likely to be generated by the same script, and hence the same attacker (or botnet). For all the queries matched by a regular expression, we look at the behavior of each IP address that issued the queries. If most of the IP addresses that issued these queries exhibit similar behavior, then it is likely that all these IPs were running the same script. We pick the following four features that are representative of querying behavior:

- *User agent:* This string contains information about the browser and the version used.
- *Metadata:* This field records certain metadata that comes with the request, e.g., where the search was issued from.

Some botnets use a fixed user agent string or metadata, or choose from a set of common values. For each group, we check the percentage of IP addresses that have identical values or identical behavior, e.g., changing value for each request. If over 90% of the IPs show similar behavior, we infer that IPs in this group might have used the same script.

- *Pages per query:* This records the number of search result pages retrieved per query.
- *Inter-query interval:* This denotes the time between queries issued by the same IP.

Queries generated by the same script may retrieve a similar number of result pages per query or have a similar inter-query interval. For these two features, we compute median value for each IP address and then check whether there is only a small spread in this value across IP addresses ($< 20\%$). This allows us to infer whether the different IPs follow the same distribution, and so belong to the same group.

Table 7 and Table 8 show the comparison between malicious queries and regular query groups. We see that for query groups returned by SearchAudit, a significant fraction of the queries agree on the metadata feature. For regular users, one usually observes a wide distribution of metadata. We see a similar trend in the user-agent string as well. For regular users, the user-agent strings rarely match, while for suspicious queries, more than half of them share the same user-agent string. With respect to the number of pages retrieved per search query, we see that regular users typically take only the first page returned. On the other hand, groups captured by SearchAudit fetch on average around 15 pages per query. This varies quite a bit across groups, with many groups fetching as few as 5 pages per query, and several groups fetching as many as 100 pages for a single query.

The average inter-query interval for normal users is over 2.5 hours between successive queries. On the other hand, the average inter-query interval for bot queries is only 7 seconds, with most of the attackers submitting the queries every second or two. A few stealthy attackers repeated search queries at a much slower rate of once every 3 minutes.

For each regular expression group, we sum up the botnet features that it matches. Figure 3 shows the distribution. A majority (87%) of the groups have at least one similar botnet feature and 69% of them have two or more features, suggesting that the queries captured by SearchAudit are probably generated by the same script.



| Field | Fraction of Queries within a Group with Same Value |
|---|---|
| Cookie enabled = *false* | 87.50% |
| Link clicked = *false* | 99.90% |

Table 5: The fraction of search queries within each regular expression group agreeing on the value of each field.

| Field | Fraction of Queries within a Group with Same Value |
|---|---|
| Cookie enabled = *false* | 2.70% |
| Link clicked = *false* | 14.23% |

Table 6: The fraction of search queries by normal users agreeing on the value of each field.

| Feature | Fraction of Queries within a Group with Same Value |
|---|---|
| User agent | 51.30% |
| Metadata | 87.50% |
| Pages per query | 14.82 |
| Inter-query interval | 6.98 seconds |

Table 7: The fraction of search queries within each SearchAudit regular expression group agreeing on botnet features.

| Feature | Fraction of Queries within a Group with Same Value |
|---|---|
| User agent | 4.02% |
| Metadata | 21.80% |
| Pages per query | 1.07 |
| Inter-query interval | 9275.5 seconds |

Table 8: The fraction of search queries by normal users agreeing on botnet features.

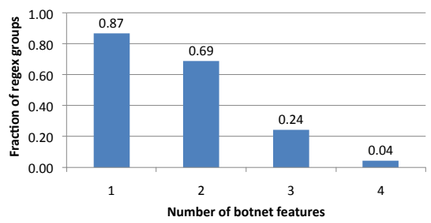



Figure 3: Graph showing the fraction of regular expressions that match one or more botnet features.

### 4.5 Discussion

Network security can be an arms race and the generated regular expressions can become obsolete [20]. However, we believe that the signature-based approach is still a viable solution, especially if we have good seed queries. In the paper, we show that even a few hundred seed queries can help identify millions of malicious queries. In addition, SearchAudit can also identify new hackers' forums or security bulletins that can be used as additional sources for seed queries. As long as there are a few IP addresses participating in different types of attacks, the query expansion framework of SearchAudit can be used to follow attackers and capture new attacks.

With the publication of the SearchAudit framework, attackers may try to work around the system and hide their activities. Attackers may try to mix the malicious searches with normal user traffic to trick SearchAudit to conclude that they are using proxy IP addresses. This is hard because behavior profiling requires attackers to submit queries that are location sensitive and also time sensitive. As many attackers use botnets to hide themselves, their IP addresses are usually spread all over the world, making it a challenging task to come up with normal user queries in all regions. In addition, as we mentioned in Section 3, proxy elimination is an optimization and it can be disabled. In such settings, both the normal queries and malicious queries can generate regular expressions. But the regular expressions of normal queries will be discarded because they match many other queries from normal users.

Attackers may also try to add randomness to the queries to escape regular expression generation. The regular expression engine looks at frequently occurring keywords to form the basis of the regular expression. Therefore, even if one attacker can manage not to reuse keywords for multiple queries, he has no control over other attackers using a similar query with the same keyword. An attacker may also simply avoid using a keyword, but since the query needs to be meaningful in order to get relevant search results, this approach would not work.

In this work, we use the presence of old cookies to help us choose regular expressions that are more likely to be malicious; old cookies are a feature associated with normal benign users. We use the cookies as a marker for normal users because it is very simple, and works well in practice. If the attackers evolve and start to use old cookies, possibly by hijacking accounts of benign users, we can rely on other features such as the presence of a real browser, long user history, actual clicking of search results, or other attributes such as user credentials.

Even if a particular attacker is very careful and manages to escape detection, if other attackers are less careful and use similar queries and get caught by SearchAudit, the careful attacker should still be found.

## 5 Stage 2: Analysis of Detection Results

In this section, we move on to the second stage of SearchAudit: analyzing malicious queries and using search to study the correlation between attacks.

The detected suspicious queries were submitted from more than 42,000 IP addresses across the globe. Large countries such as USA, Russia, and China are responsible for almost half the IPs issuing malicious queries. Looking at the number of queries issued from each IP, we find a large skew: 10% of the IPs are responsible for 90% of the queries.

SearchAudit generates around 200 regular expressions. Table 9 lists ten example regular expressions, ordered by their scores. As we can see, the lower the score, the more specific the regular expression is. The last one `.{1,25}comment.{2,21}` is an example of a discarded regular expression, with a score 0.78. It is very generic (searching for string `comment` only) and hence may cause many false positives.

By inspecting the generated regular expressions and the corresponding query results, we identify two associated attacks: finding vulnerable Web sites and forum spamming. We describe them next.

**Vulnerable Web sites:** When searching for vulnerable servers, attackers predominantly adopt two approaches:

1. They search within the structure of URLs to find ones that take particular arguments. For example,

   `index.php?content=[^?=#+;&:]{1,10}`

   searches for Web sites that are generated by PHP scripts and take arguments (`content=`). Attackers then try to exploit these Web sites by using specially crafted arguments to check whether they have popular vulnerabilities like SQL injection.

2. They perform malicious searches that are targeted, focusing on particular software with known vulnerabilities.

   We see many malicious queries that start with `"Powered by"` followed by the name of the software and version number, searching for known vulnerabilities in some version of that software.

**Forum spamming:** The second category of malicious searches are those that do not try to compromise Web sites. Instead, they are aimed towards performing certain actions on the Web sites that are generated by a particular piece of software. The most common goal is Web spamming, which includes spamming on blogs and forums. For example, a regular expression `"/includes/joomla.php" site:.[a-zA-Z]{2,3}` searches for blogs generated by the Joomla software. Attackers may have scripts to post spam to such blogs or forums.

**Windows Live Messenger phishing:** Besides identifying malicious searches generated by attackers, SearchAudit is also useful to study malicious searches triggered by normal users. In April 2009, we noticed in our search logs a large number of queries with the keyword `party`, generated by a series of Windows Live Messenger phishing attacks [25]. We see these queries because the users are redirected by the phishing Web site to pages containing the search results for the query. Since the queries are triggered by normal users compromised by the attack, expanding the queries by IP address will not gain us any information. In this case we use SearchAudit only to generate regular expressions to detect this series of phishing attacks.

In the next three sections, we study these three attacks (compromise of vulnerable Web sites, forum spamming, and Windows Live Messenger phishing) in detail. We aim to answer questions such as how do attackers leverage malicious searches for launching other attacks, how do attacks propagate and at what scale do they operate, and how can the results of SearchAudit be used to better understand and perhaps stop these attacks in their early stages.

## 6 Attack 1: Identifying Vulnerable Web Sites

As vulnerable Web sites are typically used to host phishing pages and malware, we start with a brief overview of phishing and malware attacks before describing how malicious searches can help find vulnerable Web sites.

### 6.1 Background of Phishing/Malware Attacks

A typical phishing attack starts with an attacker searching for vulnerable servers by either crawling the Web, probing random IP addresses, or searching the Web with the help of search engines. After identifying a vulnerable server and compromising it, the attacker can host malware and phishing pages on this server. Next, the attacker advertises the URL of the phishing or malware page through spam or other means. Finally, if users are tricked into visiting the compromised server, the attacker can conduct cyber crimes such as stealing user credentials and infecting computers.



| Regular Expression | Score |
|---|---|
| `"/includes/joomla\.php" site:\.[a-zA-Z]{2,3}` | 0.06 |
| `"/includes/class_item\.php" site:[^?=#+@;&:]{2,4}` | 0.08 |
| `"php-nuke" site:[^?=#+@;&:]{2,4}` | 0.16 |
| `"modules\.php\?op=modload" site:\.[a-zA-Z0-9]{2,6}` | 0.16 |
| `"[^?=#+@;&:]{0,1}index\.php\?content=[^?=#+@;&:]{1,10}` | 0.24 |
| `"powered by xoopsgallery" [^?=#+@;&:]{0,23}site:[a-zA-Z]{2,3}` | 0.30 |
| `"[^?=#+@;&:]{0,12}\?page=shop\.browse".{0,9}` | 0.35 |
| `.{0,8}index\.php\?option=com_.{3,17}` | 0.40 |
| `[^?=#+@;&:]{0,3}webcalendar v1\..{3,17}` | 0.43 |
| `.{1,25}comment.{2,21}` | 0.78 |

Table 9: Example regular expressions and their scores. The last row is an example of a regular expression that is not selected because it is not specific enough.

Currently, phishing and malware detection happens only after the attack is live, e.g., when an anti-spam product identifies the URLs in the spam email, when a browser captures the phishing content, or when anti-virus software detects the malware or virus. Once detected, the URL is added to anti-phishing blacklists. However, it is highly likely that some users may have already fallen victim to the phishing scam by the time the blacklists are updated.

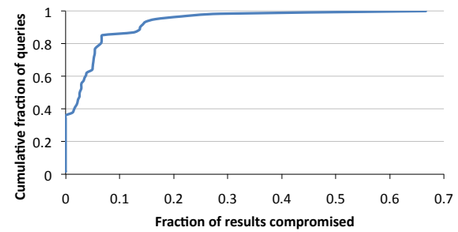



Figure 4: The fraction of search results that were present in phishing/malware feeds for each query.

### 6.2 Applications of Vulnerability Searches

With SearchAudit, we can potentially detect phishing/malware attack at the very first stage, when the attacker is searching for vulnerabilities. We might even proactively prevent servers from getting compromised.

To obtain the list of vulnerable Web sites, we sample 5,000 queries returned by SearchAudit. For every query $q$ we issue a query "$q$ -dork -vulnerability" to the search engine and record the returned URLs. Here we explicitly exclude the terms "dork" and "vulnerabilities" because we do not want results that point to security forums or hacker Web sites that discuss and post the vulnerability and the associated "dork". Using this approach, we obtain 80,490 URLs from 39,475 unique Web sites.

Ideally, we would like to demonstrate that most of these Web sites are vulnerable. Since there does not exist a complete list of vulnerable Web sites to compare against, we use several methods for our validation. First, we compare this list and a list of random Web sites against a list of known phishing or malware sites, and show that the sites returned by SearchAudit are more likely to appear in phishing or malware blacklists. Second, we test and show that many of these sites indeed have SQL injection vulnerabilities.

#### 6.2.1 Comparison Against Known Phishing and Malware Sites

For the potentially vulnerable Web sites obtained from the malicious queries, we check the presence of these URLs in known anti-malware and anti-phishing feeds. We use two blacklists: one obtained from PhishTank [2] and the other from Microsoft. In addition, we submit these queries to the search engine again at the time of our experiments in order to obtain the latest results.

In both cases, the results are similar: 3-4% of the domains listed in the search results of malicious queries are in the anti-phishing blacklists, and 1.5% of them are in the anti-malware blacklist. In total, around 5% of the domains appear in one or more blacklists. This is significantly higher than other classes of Web sites we considered.

Not all malicious queries may be equally good at finding vulnerable servers. Figure 4 shows the distribution of compromised search results across queries. For the top 10% of the queries, at least 15% of the search results appear in the blacklists.

#### 6.2.2 SQL Injection Vulnerabilities

Next, we show that a subset of these Web sites do indeed have vulnerabilities. Given that SQL injection is a popular attack, since many Web sites use database backends, we test for SQL vulnerabilities.

The best way to prove that a server has SQL injection vulnerabilities would be to actually compromise the server; however, we were not comfortable with doing this. Instead, we limit ourselves to checking if the inputs appear to be sanitized by performing the following study. For the malicious queries, we look at the search results and crawl all of the links twice. For each link, the first time we crawl the link as is, and the second time we add a single quote (') to the first argument to test whether the server sanitizes the argument correctly. Note that we consider URLs that take an argument. We then compare the Web pages obtained from the successive crawls. If the two pages are identical, then it suggests that the input arguments are being properly sanitized, so there is no obvious SQL injection vulnerability. However, if the pages are different, it does not necessarily mean that the input is not being sanitized—it could just be an advertisement that changes with each access. Instead, we look at the `diff` between the two pages, and check whether the second page contains any kind of SQL error. If there is an SQL error in the second page, but not in the first, it shows that the input string is not being filtered properly. While the presence of unsanitized inputs does not guarantee SQL injection vulnerabilities, it is nevertheless a strong indicator.

We examine a sample of 14,500 URLs obtained from the results of malicious queries, and find that 1,760 URLs (12%) do not sanitize the input strings and therefore may be vulnerable to SQL injection. Note that this is a conservative estimate since these URLs only account for Web sites that take arguments in the URL. Other Web sites that take `POST` arguments or have input forms on their pages could also be susceptible to SQL injection attacks.

## 7 Attack 2: Forum-Spamming Attacks

Using the seed queries from `milw0rm` (which were for the purpose of finding vulnerable Web sites), SearchAudit additionally identifies forum-spamming attacks. In this section, we study the forum-spamming searches in detail.

| Dataset | Forum-Searching IPs | Total Searches |
|---|---|---|
| February 2009 | 22,466 | 5,828,704 |
| December 2009 | 20,309 | 1,130,337 |
| January 2010 | 31,071 | 567,445 |

Table 11: Stats on forum-searching IPs.

### 7.1 Attack Process

Forum spamming is an effective way to deliver spam messages to a large audience. In addition, it may be used as a technique to boost the page rank of Web sites. To do so, spammers insert the URL of the target Web site that they want to promote in a spam message. By posting the message in many online forums, the target Web site would have a high in-degree of links, possibly resulting in a high page rank.

While there are several studies on the effect of forum spamming [19, 22], this section focuses on exploring the ways spammers perform forum spamming. In particular, we show how they discover a large number of forum pages in the first place.

Table 10 shows a few example forum-related queries captured by SearchAudit. There are two types of queries: the first being general like "post a new topic", and the second being more specific, tailored for a particular piece of software. For example,"`UBBCode:!JoomlaComment`" searches for pages generated by the JoomlaComment software. For both types of queries, random keywords are added to increase the search coverage. The randomness is especially useful if spammers use botnets, as each bot will get different query results and they can focus on spamming different forums in parallel.

### 7.2 Attack Scale

From the regular expressions generated by SearchAudit, we manually identified 46 regular expressions that are associated with forum spamming. Using these regular expressions, we proceeded to study the matched queries and IP addresses. Table 7.2 shows that the number of IPs used for forum searching stayed quite constant in 2009, but in 2010, the number of IP addresses increased by 50%.

Most IPs have transient behavior. Comparing the IPs in December 2009 to those in January 2010, only 3115 (10-15%) IPs overlap. This shows that the forum-spamming hosts either change frequently, or may reside on dynamic IP ranges and hence their IPs change over time. Both these possibilities suggest that they are likely to be botnet hosts. In fact, when we apply the group similarity tests to check botnet behavior (defined in Section 4.4.2), all forum groups have at least one group similarity features.



| Regular Expression | # of IPs | Group Similarity Features | Targeted Forum Generation Software |
|---|---|---|---|
| [^?=#+@;&:]{2,7} "Commenta" !JoomlaComment -""#R# | 253 | 3 | Joomla |
| [^?=#+@;&:]{6,11} "ips, inc" | 9159 | 4 | IP.Board |
| [^?=#+@;&:]{1,8} "Message:" photogallery#R# | 253 | 3 | PhotoPost |
| [^?=#+@;&:]{1,9} "Be first to comment this article" akocomment#R# | 255 | 4 | AkoComment |
| [^?=#+@;&:]{1,6} "UBBCode:" !JoomlaComment -""#R# | 255 | 3 | JoomlaComment |
| [^?=#+@;&:]{1,8} "The comments are owned by the poster\. We aren't responsible for their content\." sections#R# | 253 | 3 | PHP-Nuke, Xoops, etc. |
| [a-zA-Z]{4,12} post new topic | 1028 | 1 | phpBB, Gallery, etc |
| [^?=#+@;&:]{5,13} Board Statistics.{0,10} | 302 | 1 | Invision Power Board (IP.Board), MyBB, etc. |
| BBS [a-zA-Z]{4,12} | 1861 | 1 | Infopop etc. |
| yabb [a-zA-Z]{4,14} | 388 | 1 | yaBB |
| ezboard [a-zA-Z]{4,11} | 388 | 1 | ezboard |
| VBulletin [a-zA-Z]{4,11} | 360 | 1 | Vbulletin |

Table 10: Example regular expressions related to forum searches, their scale, and the targeted forum generation software.

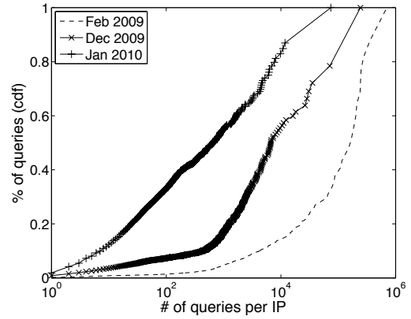



Figure 5: CDF of the distribution of queries among IPs based on the query volume.

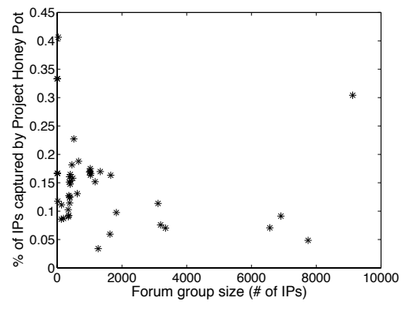



Figure 6: Fraction of IP addresses appearing in the *Project Honey Pot* list vs. the forum group size.

It is interesting to note that, although the number of IPs increased, the total number of queries decreased. As shown in Figure 5, IPs are becoming more stealthy. In February 2009, more than 80% of forum queries were originated from very aggressive IPs that submitted thousands of queries per IP. Those IPs could be spammers' own dedicated machines. In Jan 2010, less than 20% of forum queries are from aggressive IPs. The majority of the queries are from IPs that search at a low rate.

### 7.3 Applications of Forum Searching Queries

Knowledge of forum-searching IPs and query search terms can be used to help filter forum spam. After a malicious search, we can follow the search result pages to clean up the spam posts. More aggressively, even before the malicious search, by recognizing the malicious query terms or the malicious IP addresses, search engines could refuse to return results to the spammers. Web servers could also refuse connections from IPs that are known to search for forums.

We validate the forum-spamming IPs using Project Honey Pot [3]. Project Honey Pot is a distributed honeypot network that aims to identify Web spamming. Participating Web sites embed a piece of software that dynamically generates a page containing a different email address for each HTTP request. Requests are recorded and the generated email addresses are also monitored. If later they receive emails (which must be spam, since these email addresses are unused), Project Honey Pot

will know which IP addresses obtained those email addresses, and which IP addresses sent the spam emails.

Around 12% of the forum searching IPs found by SearchAudit were captured by Project Honey Pot. In contrast, among IP addresses that conduct normal queries such as `craigslist`, only 0.5% of them were listed. This shows that the captured forum searching IPs have a much higher chance of being caught spamming than the IP addresses of normal users.

Figure 6 plots the matching percentages of different regular expression groups related to forum searching. We can see that, across different groups, the percentages of forum IPs appeared in Project Honey Pot are all significant. This suggests that most of the forum-spamming groups are involved in email address scraping as well. For the largest forum-spamming group, which has 9125 IP addresses, more than 30% of the IP addresses appeared in Project Honey Pot. It is possible that the remaining 70% are also associated with spamming, but they could have targeted Web sites that are not part of their network, and are hence not captured. Hence, the analysis of search logs complements Project Honey Pot. It offers a unique view that allows us to observe all the IP addresses conducting forum searches, while Project Honey Pot allows us to see what the attackers do after performing the searches.

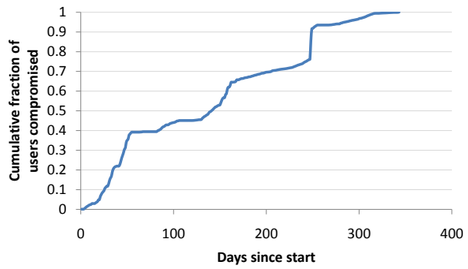



Figure 7: The rate at which new users were compromised by the phishing attack.

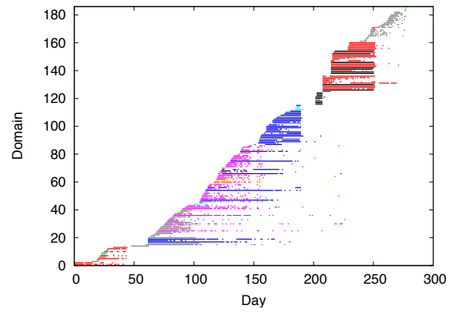



Figure 8: The timeline of how different domain names were used during the phishing attack. All lines of the same color correspond to the same IP address.

## 8 Attack 3: Windows Live Messenger Phishing Attacks

In this section, we study a series of Windows Live Messenger phishing attacks. The queries were not issued by attackers directly. Rather, they were triggered by normal users. In this section, we use SearchAudit to generate regular expressions and study this series of attacks.

### 8.1 Attack Process

The scheme of these phishing attacks operates as follows:

1. The victim (say Alice) receives a message from one of her contacts, asking her to check out some party pictures, with a link to one of the phishing sites.
2. Alice clicks the link and is taken to the Web page that looks very similar to the legitimate Windows Live Messenger login screen and asks her to enter her messenger credentials. Alice enters her credentials.
3. Alice is now taken to a page `http://<domain-name>.com?user=alice`, which redirects to image search results from a search engine (in this case, Bing) for `party`.
4. The attackers now have Alice's credentials. They log in to Alice's account and send a similar message to her friends to further propagate the attack.

We believe there are two reasons why the attackers use a search engine here. First, using images from a search engine is less likely to tip the victim off than if the images were hosted on a random server. Second, the attackers do not need to host the image Web pages themselves, and can thus offload the cost of hosting to the search engine servers.

### 8.2 Attack Scale

Since this attack generated search traffic that contains the keyword `party`, we feed this keyword as the seed query into SearchAudit. Since all the queries of this attack are identical or similar, we modify SearchAudit to focus on the query referral field, which records the source of traffic redirection. SearchAudit generates two regular expressions from the query referral field:

`1.http://`**`[a-zA-Z0-9._]*`**`.<domain-name>/`
`2.http://<domain-name>?user=`**`[a-zA-Z0-9._]*`**



In the second regular expression, the pattern `[a-zA-Z0-9._]*` may seem like a random set of letters and numbers, but it actually describes usernames. In our example attack scenario, when Alice is redirected to the image search results, the HTTP-referrer is set to `http://<domain-name>.com?user=alice`. Using this information, we can identify the set of users whose credentials may have been compromised.

Using these regular expressions, SearchAudit identifies a large number of unique user names in the log collected from May 2008 to July 2009. Figure 7 shows the cumulative fraction of users compromised by this attack over time. When the attack first started, there was an exponential growth phase, similar to other worm or virus breakouts. This phase ended around day 50, when most of the domains got blacklisted (see Figure 8). This attack then transited into a steady increase phase, until day 250 when it broke out again.

There are over 400 unique phishing domain names associated with this attack. The top domains targeted more than $10^5$ users. Around one third of the domains phished fewer than 100 users each. These domains were the ones that were quickly blacklisted. Figure 8 plots the timeline of how different domains were used over time. For readability, the plot contains only the top domains (out of the total 400 domains) that were responsible for compromising at least 1000 users. The figure plots the domains on the Y-axis, and the days on which that domain was active on the X-axis. Each horizontal line corresponds to the set of days a particular domain was seen in our search log. The different colors correspond to the different IP addresses on which the Web pages were hosted. We observe that though there were over 180 domain names in circulation, they were all hosted on only a dozen different IP addresses. It can also be seen that multiple domain names were associated with an IP address at the same time. Therefore, it is not the case that a new domain name was registered and used only after an older one was blocked.

### 8.3 Characteristics of Compromised Accounts

We find that the compromised accounts had a large number of short login sessions (lasting less than one minute). These short login sessions were initiated from IPs in several different `/24` subnets. Figure 9 shows the comparison between the short logins from multiple subnets for compromised users and for the other users. We see that for typical users, 99% of the short logins happened from fewer than 4 different subnets. However, for the compromised users, we see that more than 50% had short logins from 15 or more different subnets.

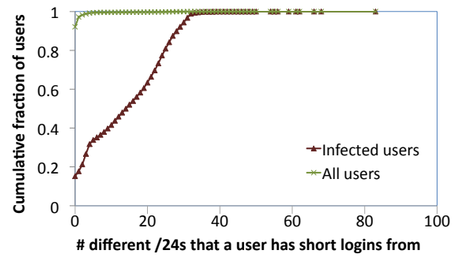



Figure 9: Number of different `/24` subnets from which short logins happen.

We also observe that many of the short logins came from IPs which were located in Hong Kong. Given that the phishing sites were also mostly located in Hong Kong, the attackers might have resources in Hong Kong, where they logged in to the compromised accounts and sent messages to spread the phishing attacks.

Using these characteristics, we can then look back at the login patterns of all Windows Live Messenger users to identify more user accounts with similar suspicious login patterns, thus enabling us to take remedial actions for protecting a larger number of compromised users.

## 9 Conclusion

In this paper we present SearchAudit, a framework to identify malicious Web searches. By taking just a small number of known malicious queries as seed, SearchAudit can identify millions of malicious queries and thousands of vulnerable Web sites. Our analysis showes that the identification of malicious searches can help detect and prevent large-scale attacks, such as forum spamming and Windows Live Messenger phishing attacks. More broadly, our findings highlight the importance of analyzing search logs and studying correlations between the various attacks enabled by malicious searches.

## Acknowledgements

We thank Fritz Behr, Dave DeBarr, Dennis Fetterly, Geoff Hulten, Nancy Jacobs, Steve Miale, Robert Sim, David Soukal, and Zijian Zheng for providing us with data and feedback on the paper. We are also grateful to anonymous reviewers for their valuable comments.

## References

[1] milw0rm.com. `http://www.milw0rm.com/`.

[2] PhishTank - Join the fight against phishing. `http://www.phishtank.com`.

[3] Project Honey Pot. http://www.projecthoneypot.org/home.php.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[5] G. Buehrer, J. W. Stokes, and K. Chellapilla. A large-scale study of automated Web search traffic. In *the 4th International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, 2008.

[6] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *the 12th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[7] N. Daswani and M. Stoppelman. The anatomy of Clickbot.A. In *the 1st Conference on Hot Topics in Understanding Botnets (HotBots)*, 2007.

[8] E. N. Efthimiadis, N. Malevris, A. Kousaridas, A. Lepeniotou, and N. Loutas. An evaluation of how search engines respond to greek language queries. In *HICSS*, 2008.

[9] D. Eichmann. The RBSE spider - Balancing effective search against Web load, 1994.

[10] S. Frantzen. Clickbot. http://isc.sans.org/diary.html?storyid=1334.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, 2007.

[12] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *the 13th Conference on USENIX Security Symposium*, 2004.

[13] C. Kreibich and J. Crowcroft. Honeycomb: Creating intrusion detection signatures using honeypots. In *the 2nd Workshop on Hot Topics in Networks (HotNets-II)*, 2003.

[14] B. W. Lampson. Computer security in the real world. *IEEE Computer*, 37(6):37–46, June 2004.

[15] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worm with provable attack resilience. In *IEEE Symposium on Security and Privacy*, 2006.

[16] T. Moore and R. Clayton. Evil searching: Compromise and recompromise of Internet hosts for phishing. In *13th International Conference on Financial Cryptography and Data Security*, 2009.

[17] H. Moukdad. Lost in cyberspace: How do search engines handle Arabic queries. In *the 32nd Annual Conference of the Canadian Association for Information Science*, 2004.

[18] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, 2005.

[19] Y. Niu, Y. Wang, H. Chen, M. Ma, and F. Hsu. A quantitative study of forum spamming using context based analysis. In *Network and Distributed System Security (NDSS) Symposium*, 2007.

[20] N. Provos, J. McClain, and K. Wang. Search worms. In *the 4th ACM Workshop on Recurring Malcode (WORM)*, 2006.

[21] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Operating Systems Design and Implementation (OSDI)*, 2004.

[22] Y. Wang, M. Ma, Y. Niu, and H. Chen. Spam double-funnel: Connecting Web spammers with advertisers. In *World Wide Web Conference (WWW)*, 2007.

[23] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: Signatures and characteristics. In *SIGCOMM*, 2008.

[24] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *the 14th USENIX Security Symposium*, 2005.

[25] F. Yu, Y. Xie, and Q. Ke. Sbotminer: Large scale search bot detection. In *International Conference on Web Search and Data Mining (WSDM)*, 2010.

[26] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Operating Systems Design and Implementation (OSDI)*, 2008.

# Toward Automated Detection of Logic Vulnerabilities in Web Applications

Viktoria Felmetsger Ludovico Cavedon Christopher Kruegel Giovanni Vigna
*[rusvika,cavedon,chris,vigna]@cs.ucsb.edu*
*Computer Security Group*
*Department of Computer Science*
*University of California, Santa Barbara*

## Abstract

Web applications are the most common way to make services and data available on the Internet. Unfortunately, with the increase in the number and complexity of these applications, there has also been an increase in the number and complexity of vulnerabilities. Current techniques to identify security problems in web applications have mostly focused on input validation flaws, such as cross-site scripting and SQL injection, with much less attention devoted to application logic vulnerabilities.

Application logic vulnerabilities are an important class of defects that are the result of faulty application logic. These vulnerabilities are specific to the functionality of particular web applications, and, thus, they are extremely difficult to characterize and identify. In this paper, we propose a first step toward the automated detection of application logic vulnerabilities. To this end, we first use dynamic analysis and observe the normal operation of a web application to infer a simple set of behavioral specifications. Then, leveraging the knowledge about the typical execution paradigm of web applications, we filter the learned specifications to reduce false positives, and we use model checking over symbolic input to identify program paths that are likely to violate these specifications under specific conditions, indicating the presence of a certain type of web application logic flaws. We developed a tool, called Waler, based on our ideas, and we applied it to a number of web applications, finding previously-unknown logic vulnerabilities.

## 1 Introduction

Web applications have become the most common means to provide services on the Internet. They are used for mission-critical tasks and frequently handle sensitive user data. Unfortunately, web applications are often implemented by developers with limited security skills, who often have to deal with time-to-market pressure and financial constraints. As a result, the number of web application vulnerabilities has increased sharply. This is reflected in the Symantec Global Internet Security Threat Report, which was published in April 2009 [12]. The report states that, in 2008, web vulnerabilities accounted for 63% of the total number of vulnerabilities reported.

Most recent research on vulnerability analysis for web applications has focused on the identification and mitigation of input validation flaws. This class of vulnerabilities is characterized by the fact that a web application uses external input as part of a sensitive operation without first checking or sanitizing it properly. Prominent examples of input validation flaws are cross-site scripting (XSS) [20] and SQL injection vulnerabilities [3, 32]. With XSS, an application sends to a client output that is not sufficiently checked. This allows an attacker to inject malicious JavaScript code into the output, which is then executed on the client's browser. In the case of SQL injection, an attacker provides malicious input that alters the intended meaning of a database query.

One reason for the prior focus on input validation vulnerabilities is that it is possible to provide a concise and general specification that captures the essential characteristics of these vulnerabilities. That is, given a programming environment, it is possible to specify a set of functions that read inputs (called *sources*), a set of functions that represent security-sensitive operations (called *sinks*), and a set of functions that check data for malicious content. Then, various static and dynamic analysis techniques can be used to ensure that there are no unchecked data flows from sources to sinks. Since the specification of input validation flaws is independent of the application logic, once a detection system is available, it can be used to find bugs in many applications.

While it is important to identify and correct input validation flaws, they represent only a subset of the spectrum of (web application) vulnerabilities. In this paper, we explore another type of application flaws. In particular, we look at vulnerabilities that result from errors in the logic

of a web application. Such errors are typically specific to a particular web application, and might be domain-specific. For example, consider an online store web application that allows users to use coupons to obtain a discount on certain items. In principle, a coupon can be used only once, but an error in the implementation of the application allows an attacker to apply a coupon an arbitrary number of times, reducing the price to zero.

So far, web application logic flaws have received little attention, and their treatment is limited to informal discussions (a well-known example is the white paper by J. Grossman [14]). This is due to the fact that logic vulnerabilities are specific to the intended functionality of a web application. Therefore, it is difficult (if not impossible) to define a general specification that allows for the discovery of logic vulnerabilities in different applications.

One possible approach would be to leverage an application's requirement specification and design documents to identify parts of the implementation that do not respect the intended behavior of the application. Unfortunately, these documents are almost never available in the case of web applications. Therefore, other means to characterize the expected behavior of web application must be found for detection of application logic flaws.

In this paper, we take a first step toward the automated detection of application logic vulnerabilities. Our approach operates in two steps. In the first step, we infer specifications that (partially) capture a web application's logic. These specifications are in the form of likely invariants, which are derived by analyzing the dynamic execution traces of the web application during normal operation. The intuition is that the observed, normal behavior allows one to model properties that are likely intended by the programmer. This step is necessary to automatically obtain specifications that reflect the business logic of a particular web application. In the second step, we analyze the inferred specifications with respect to the web application's code and identify violations.

The current implementation of our approach is based on two well-known analysis techniques, namely, dynamic execution to extract (likely) program invariants and model checking to identify specification violations. However, to the best of our knowledge, the way in which we combine these two techniques is novel, has never been applied to web applications, and has not been leveraged to detect application logic flaws. Moreover, we had to significantly extend the existing techniques to capture specific characteristics of web applications and to scale them to real-world applications as outlined below.

In the first step of our analysis, we used a well-known dynamic analysis tool [9, 11] to infer program specifications in the form of likely invariants. We extended the existing general technique to be more targeted to the execution of web applications. In particular, we addressed two main shortcomings of the general approach: the fact that many invariants that relate to important concepts of web applications were not identified (e.g., invariants related to objects that are part of the user session) and the fact that many spurious invariants were generated as a result of the limited coverage of the dynamic analysis step or because of artifacts in the analyzed inputs.

To deal with spurious invariants, we developed two novel techniques to identify which derived invariants reflect real (or "true") program specifications. The first one uses the presence of explicit program checks, involving the variable(s) constrained by an invariant, as a clue that the invariant is indeed relevant to the behavior of the web application. The second one is based on the idea that certain types of invariants are intrinsically more likely to reflect the intent of the programmer. In particular, we focus on invariants that relate external inputs to the contents of user sessions and the back-end database. The use of these techniques to filter the derived invariants allows for a more effective extraction of specification of a web application's behavior, when compared to previously-proposed approaches that accept all generated likely invariants as correctly reflecting the behavior of a program.

In the second step of the analysis, we use model checking over symbolic input to analyze the inferred specifications with respect to the web application's code and to identify which real invariants can be violated. We had to extend existing model checking tools with new mechanisms to take into account the unique characteristics of web applications. These characteristics include the fact that web applications are composed of modules that can be invoked in any order and that the state of the web application must also take into account the contents of back-end databases and other session-related storage facilities.

By following the two steps outlined above, it is possible to automatically detect a certain subclass of application logic flaws, in which an application has inconsistent behavior with respect to security-sensitive functionality. Note that our approach is neither sound nor complete, and, therefore, it is prone to both false positives and false negatives. However, we implemented our approach in a prototype tool, called Waler, that is able to automatically identify logic flaws in web applications based on Java servlets. We applied our tool to several real-world web applications and to a number of student projects, and we were able to identify many previously-unknown web application logic flaws. Therefore, even though our technique cannot detect all possible logic flaws and our tool is currently limited to servlet-based web applications, we believe that this is a promising first step towards the automated identification of logic flaws in web applications.



In summary, this paper makes the following contributions:

- We extend existing dynamic analysis techniques to derive program invariants for a class of web applications, taking into account their particular execution paradigm.
- We identify novel techniques for the identification of invariants that are "real" with high probability and likely associated with the security-relevant behavior of a web application, pruning a large number of spurious invariants.
- We extend existing model checking techniques to take into account the characteristics of web applications. Using this approach, we are able to identify the occurrence of two classes of web application logic flaws.
- We implemented our ideas in a tool, called Waler, and we used it to analyze a number of servlet-based web applications, identifying previously-unknown application logic flaws.

## 2 Web Application Logic Vulnerabilities

Web application vulnerabilities can be divided into two main categories, depending on how a vulnerability can be detected: (1) vulnerabilities that have common characteristics across different applications and (2) vulnerabilities that are application-specific. Well-known vulnerabilities such as XSS and SQL injection belong to the first category. These two vulnerabilities are characterized by the fact that a web application uses external input as part of a sensitive operation without first checking or sanitizing it. Vulnerabilities of the second type (such as, for example, failures of the application to check for proper user authorization or for the correct prices of the items in a shopping cart) require some knowledge about the application logic in order to be characterized and identified. In this paper, we focus on this second type of vulnerabilities, and we call them *web application logic vulnerabilities*.

To detect web application logic vulnerabilities automatically, one needs to provide the detection tool with a specification of the application's intended behavior. Unfortunately, these specifications, whether formal or informal, are rarely available. Therefore, in this work, we propose an automated way to detect application logic vulnerabilities that do not require the specification of the web application behavior to be available. Our intuition is that often the application code contains "clues" about the behavior that the developer intended to enforce. These "clues" are expressed in the form of constraints on the values of variables and on the order of the operations performed by the application.

There are many ways in which constraints can be implemented in an application. In this work, we focus on two concrete types of constraints. The first (and most intuitive) way to encode application-specific constraints is in the form of program checks (i.e., *if*-statements). The presence of such a check in the program before certain data or functionality is accessed often represents a "clue" that either the range of the allowed input should be limited or that an access to an item is limited. The absence of a similar check on an alternate program path to the same program point might represent a vulnerability. For example, vulnerabilities like authentication bypass, where an attacker is able to invoke a privileged operation without having to provide the necessary credentials, could be detected using this approach.

The second type of constraints, which often exist in web applications, is the implicit correlation between the data stored in back-end databases and the data stored in user sessions. More specifically, in web applications, databases are often used to store persistent data, and user sessions are used to store the most accessed parts of this data (such as user credentials). Thus, there often exist implicit constraints on what is currently stored in the user session when a database query is issued. A "clue," in this case, is an explicit relation between session data and database data. Certain application logic vulnerabilities, like unauthorized editing of a post belonging to another user, can be detected if a path where these relations are violated is found. More detailed examples of this type of vulnerabilities will be provided in Section 4.3.2.

## 3 Detection Approach

Based on the discussions in the previous section, it is clear that an analysis tool that aims to detect web application logic vulnerabilities requires a specification of expected behavior of the program that should be checked. If such specifications are available (e.g., in the form of formal specifications or unit testing procedures), they can be leveraged to validate the behavior of the application's implementation. However, in many cases there is no specification of the expected behavior of a web application. In these cases, we need a way to derive it in an automated fashion.

A number of techniques has been proposed by various researchers to derive program specification automatically. However, regardless of the approach used, none of them can derive a complete specification without human feedback. To overcome this problem, we propose to use one of the existing dynamic techniques to derive partial program specifications and use an additional analysis step to refine the results and find vulnerabilities.

In particular, we observe that web applications are typically exercised by users in a way that is consistent with

the intentions of the developers. More specifically, users usually browse the application by following the provided links and filling out forms with expected input. These program paths are usually well-tested for normal input. As a result, when monitoring a web application whose "regular" functionality is exercised, it is possible to infer interesting relationships between variables, constraints on inputs and outputs, and the order in which the application's components are invoked. This information can be used to extract specifications that partially characterize the intended behavior of the web application.

As a result, in our approach, we use an initial dynamic step where we monitor the execution of a web application when it operates on a number of normal inputs. In this step, it is important to exercise the application functionality in a way that is consistent with the intentions of the developer, i.e., by following the provided links and submitting reasonable input. Note that the information about a web application's "normal" behavior cannot be gathered using automatic-crawling tools, as these tools usually do not interact with an application following the workflow intended by the developer or using inputs that reflect normal operational patterns.

In this work, as the result of the dynamic analysis step, we infer partial program specifications in the form of likely invariants. These invariants capture constraints on the values of variables at different program points, as well as relationships between variables. For example, we might infer that the Boolean variable `isAdmin` must be true whenever a certain (privileged) function is invoked. As another example, the analysis might determine that the variable `freeShipping` is true only when the number of items in the shopping cart is greater than 5. We believe that these invariants provide a good base for the detection of logic flaws because they often capture application-specific constraints that the programmer had in mind when developing the web application. Of course, it is unlikely that the set of inferred invariants represents a complete (or precise) specification of a web application's functionality. Nevertheless, it provides a good, initial step to obtain a model of the intended behavior of a program and can be used to guide further, more elaborate program analysis.

As the second step of the analysis, we use model checking with symbolic inputs to check the inferred specifications. The goal is to find additional evidence in the code about which invariants are likely to be part of the real program specification and then to identify paths where these invariants are violated.

A naïve approach would assume that all the generated invariants represent real invariants (specifications) for an application. Unfortunately, this straightforward solution leads to an unacceptably large number of false positives. The reason is the incompleteness of the dynamic analysis step. In particular, the limited variety of the input data frequently leads to the discovery of spurious invariants that do not reflect the intended program specification. To address this problem, we propose two novel techniques to distinguish between spurious and real program invariants.

The first technique aims to distinguish between a spurious and a true invariant by determining whether a program contains a *check* that involves the variables contained in the invariant on a path leading to the program point for which this likely invariant was generated. A check on a variable is a control flow operation that constrains this variable on a path. For example, the *if*-statement `if (isAdmin == true) {...}` represents a check on the variable `isAdmin`. Intuitively, we assume that a certain invariant was intended by a programmer if there is at least one program path that contains checks that enforce the correctness of this invariant (i.e., the checks imply that the invariant holds). We call such invariants *supported invariants*. When we find a supported invariant that can be violated on an alternative program path leading to the same program point, we report this as a potential application logic vulnerability. When a likely invariant can be violated, but there are no checks in the program that are related to this invariant, then we consider it to be spurious.

The second technique identifies a certain type of invariant that we always consider to reflect actual program specifications. These invariants represent equality relations between web application state variables (in particular, variables storing the content of user sessions and database contents). Relationships of that kind often reflect important internal consistency constraints in a web application and are rarely coincidental. A vulnerability is reported when the analysis determines that the equality relation is not enforced on all paths.

The vulnerability detection process and our techniques to distinguish between spurious and real invariants are discussed in more detail in Section 4.3.

# 4 Implementation

We chose to implement the proposed approach for servlet-based web applications written in Java. Servlets are frequently used for implementing web applications. In addition, there are a number of existing tools available for Java that can be used for program analysis. In this section, we describe the tools that we used, the extensions that we developed, and the challenges that we had to overcome to make them work together.

We first briefly introduce servlets [24]. A typical servlet-based web application consists of servlets, static documents, client-side code, and descriptive meta-information. A *servlet* is a Java-based web component



```
package myapp;
public class User {
  private String username;
  private String role;
}
public class Order {
  private int tax;
  private int total;
  private Cart cart;
}
public class Cart {
  private List products;
  private int total;
}
```

Class Definitions

```
_jspService(javax.servlet.http.HttpServletRequest req,
            javax.servlet.http.HttpServletResponse res)
            :::EXIT106

   // invariants for the field "role" belonging to an
   // object stored in the session under the key "user"
   req.session.user.role != null
   req.session.user.role.toString == ``admin''

   // invariants for the fields "cart" and "total"
   // stored in the session under the key "order"
   req.session.order.cart.total
                    == req.session.order.total
   req.session.order.total > req.session.order.tax
```

Generated Invariants

Figure 1: Example of invariants generated for an exit point on line 106 of the *_jspService* method of a servlet.

whose methods are executed on the server in response to certain web requests. Servlets are managed by a *servlet container*, which is an extension of a web server that loads/manages servlets and provides services via a well-defined API. These services include receiving and mapping requests to servlets, sending responses, caching, enforcing security restrictions, etc. Servlets can be developed as Java classes or as JavaServer Pages (JSPs). JSPs are a mix of code and static HTML content, and they are translated into Java classes that implement servlets.

## 4.1 Deriving Specifications

As mentioned previously, in this work, we consider program specifications that can be expressed as invariants over program variables. To derive these invariants, we leverage Daikon [9, 11], a well-known tool for dynamic detection of likely program invariants.

**Daikon.** Daikon generates program invariants using application execution traces, which contain values of variables at concrete program points. It is capable of generating a wide variety of invariants that cover both single variables (e.g., $total \geq 50.0$) and relationships between multiple variables (e.g., $total = price * num + tax$). Daikon-generated invariants are called *likely invariants* because they are based on dynamic execution traces and might not hold on all program paths.

Daikon comes with a set of front-ends. Each front-end is specific to a certain programming language (such as C or Java). The task of a front-end is to instrument a given program, execute it, and create data trace files. These trace files are then fed to Daikon for invariant generation. For our analysis, we leveraged the existing front end for Java, called Chicory, and plugged it into a JVM on top of which the Tomcat servlet engine [13] is executed. This allowed us to intercept and instrument all servlets executed by the Tomcat server.

The current implementation of Chicory produces traces only for procedure entry and exit points and non-local variables. Therefore, Daikon generates invariants for method parameters, function return values, static and instance fields of Java objects, and global variables.

**Our changes.** In addition to altering Chicory's invocation model to work with Tomcat, we extended Chicory with a way to include the content of user sessions into the generated execution traces. Invariants over this data are important for the vulnerability analysis of web applications because user sessions are an integral part of an application's state and directly affect its logic.

The content of user sessions is stored by a servlet container in the form of dynamically-generated mappings from a key to a value, i.e., as elements in a hash map container. We found that, given the current design of Daikon and Chicory, it is not possible to generate useful invariants for the contents of such containers. The reason is that Daikon requires the type and the name of all variables that can appear at a particular program point to be declared before the first trace for a particular program point is generated. This information is not available beforehand for containers like hash maps because they are dynamically-sized and can contain elements of different types.

To generate valid traces for Daikon, Chicory generates all declarations for program points at the application loading time. At this time, it needs to know the exact type of each variable/object in declaration to be able to traverse the object structure and generate precise (or interesting) invariants. For example, in order to generate a definition for the field *role* of the object of type *User* (defined in Figure 1), which might be stored in the user session of a servlet application under the key "user," Chicory needs to know that the object of the type *User* is expected in the session.

To overcome these problems, we provide our front-end with possible mappings from a key to an object type that can be observed in a session during execution. For example, for the code shown in Figure 1, we would need to provide the following mappings:

```
user:myapp.User
cart:myapp.Cart
order:myapp.Order
```

We modified Chicory to use this information to generate more precise traces for session data. This information allows for the generation of more interesting invariants, such as the ones shown in the Figure 1. We extended the front-end to generate traces for the content of user sessions for every method in an application. As future work, we plan to generate these mapping automatically for arbitrary containers by generating new declarations as new elements are found in a container, and then merging the resulting traces before feeding them to Daikon.

To generate program execution traces, we wrote scripts to automatically operate web applications. For each application, these scripts simulate typical user activities, such as creating user accounts, logging into the application, choosing and buying items from a store, accessing administrative functionality, etc. The main idea of this step is to exercise the application's common execution paths by following the links and filling out the forms presented to the user during a typical interaction with the application. The final outcome of the dynamic analysis step is a file containing a serialized version of likely invariants for the given web application. These invariants serve as a (partial, simplified) specification of the web application, and they are provided as input to the next step of the analysis.

## 4.2 Model Checking Applications

Once the approximate specifications (i.e., the likely invariants) for a web application have been derived, the next step is to analyze the application for supporting "clues" and identify invariants that are part of a true program specification. Any violation of such an invariant represents a vulnerability.

We chose to use model checking for this step of the analysis and implemented it in a tool called Waler (Web Application Logic Errors AnalyzeR). Given a servlet-based application and a set of likely invariants, Waler systematically instantiates and executes symbolically the servlets of the application imitating the functionality of a servlet container. As the application is executed, Waler checks the truth value of provided likely invariants, analyzes the application's code for "clues," and reports possible logic errors. In this section, we describe the architecture and execution model of Waler. Then, in Section 4.3 we explain how Waler identifies interesting invariants and application logic vulnerabilities.

### 4.2.1 System Top-level Design

Waler is implemented on top of the Java PathFinder (JPF) framework [19, 35], and its general architecture is shown

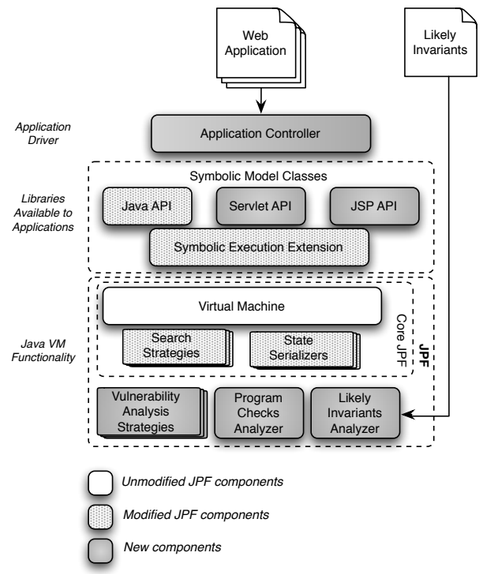



Figure 2: Waler's architecture.

Figure 2. In this figure, dark gray boxes represent new modules that we implemented, while dotted (light gray) boxes represent parts of JPF that we had to extend.

**JPF overview.** JPF is an open-source, explicit-state model checker that implements a JVM. It systematically explores an application's state space by executing its bytecode. JPF consists of a number of configurable components. For example, the specific way in which an application's state space is explored depends on a chosen *Search Strategy* – JPF core distribution includes a number of basic strategies. The *State Serializer* component defines how an application state is stored, matched against others, and restored. JPF also comes with a number of interfaces that allow for its functionality to be extended and modified in arbitrary ways.

In general, JPF is capable of executing any Java class-file that does not depend on platform-specific native code, and many of the Java standard library classes can run on top of JPF unmodified. However, in JPF, some of the Java library classes are replaced with their *model* versions to reduce the complexity of their real implementations and/or to enable additional features. For example, Java classes that have native method calls (such as file I/O) have to be replaced by their models, which either emulate the required functionality or delegate the native calls to the actual JVM on top of which JPF is executed.



Also, JPF comes with a number of extensions that provide additional functionality on top of JPF. Below, we discuss the JPF-SE extension for JPF, which we leveraged in Waler to enable symbolic execution.

**The JPF-SE Extension.** The JPF-SE extension for JPF enables symbolic execution of programs over unbounded input when using explicit-state model checking [2]. With this extension, the Java bytecode of an application needs to be transformed so that all concrete basic types, such as integers, floats, and strings, are replaced with the corresponding symbolic types. Similarly, concrete operations need to be replaced with the equivalent operations on symbolic values. For example, all objects of type *int* are replaced with objects of type *Expression*. An addition of two integers is replaced with a call to the *_plus* method of the *Expression* class. Following the standard symbolic execution approach, all newly-generated constraints are added to the *path condition* (PC) over the current execution path. The generation of constraints is done in the methods of symbolic classes, and it is transparent to the application. Whenever the PC is updated, it is checked for satisfiability with a constraint solver, and infeasible paths are pruned from execution.

Unfortunately, we found that JPF-SE was missing a considerable amount of functionality that needed to be added to make the system suitable for real-world applications. For example, the classes implementing symbolic string objects were missing a significant number of symbolic methods with respect to the *java.lang.String API*, which is used extensively in web applications. Also, in order to execute an arbitrary application using JPF-SE, symbolic versions of many standard Java libraries are required. These libraries were not provided with the extension. Finally, a tool to perform the necessary transformations of Java bytecode was not publicly available, and, therefore, we implemented our own transformer by leveraging ASM [25], a Java bytecode engineering library.

**Waler overview.** In order to execute servlet-based web applications and analyze them for logic errors, we had to extend JPF in a number of ways. As shown in Figure 2, we implemented from scratch four main components: the *Application Controller* (AC), the *Vulnerability Analysis Strategies* (VAS), the *Program Checks Analyzer* (PCA), and the *Likely Invariants Analyzer* (LIA). The AC component is responsible for loading, mapping, and systematically initiating execution of servlets in a servlet-based application. As the analyzed application itself, it runs on top of the JVM implemented by core-JPF and uses symbolic versions of Java libraries.

The other three components are internal to JPF, i.e., they are not visible to web applications and do not rely on model classes. The LIA component is responsible for parsing Daikon-generated invariants and checking their truth value as a program executes. The PCA component keeps track of all the program checks performed by an application on an execution path. Finally, the VAS component provides various strategies for vulnerability detection based on the information provided by LIA and PCA. We provide more details on how these modules work in the following sections.

In addition, we had to extend a number of existing JPF components to address the needs of our analysis. In particular, we modified existing search strategies, state information tracking, and implemented some missing parts of JPF-SE. Due to space limitations, we will not explain all of the changes unless they are significant for understanding our approach.

Finally, we extended JPF with a set of 40 model classes that provide the servlet API and related interfaces (such as the JSP API). These classes implement the standard functionality of a servlet container, but instead of reading and writing actual data from/to the network, they operate on symbolic values. Our implementation is based on the real implementation of the servlet container for Tomcat.

### 4.2.2 Execution Model

To systematically analyze a web application for logic errors, Waler needs to be able to model all possible user interactions with the application. To achieve that, it needs to find all possible entry points to the application and execute all the possible sequences of invocations using symbolic input.

In general, a user can interact with a web application in different ways: one can either follow the links (leading to URLs) presented by the application (as part of a web page) or can directly point the browser to a certain URL. On the server side, after (and if) a request URL is mapped to a servlet-based application, the path part of the URL is used to locate a particular servlet that will handle the request. We call the set of all such URL paths that lead to the invocation of a servlet the "application entry points."

Thus, before a program can be analyzed, we need to identify all possible application entry points. In the general case, there can be an infinite number of URLs that lead to an invocation of a servlet; however, for each particular application, there is a finite and well-defined number of possible mappings from a request URL pattern to a servlet. Thus, for the analysis, it is sufficient to find all such mappings. For example, if an application has the URL */login* mapped to the *AuthManager* servlet and the URLs */cart* and */checkout* mapped to the *CartManager* servlet, it can be said that the application has three entry points. In servlet-based applications, it is also possible to have wildcard mappings, such as *account/**, mapped to a servlet. In this case, all URL paths starting with */account/* are mapped to the same servlet. We consider

such mappings to represent single entry points and simply treat the part of the URL that matches the "*" as a symbolic input. This is consistent with our handling of other request parameters accessed by servlets, which are also represented by symbolic values.

To find all entry points, our system inspects the application deployment descriptor (typically, the *web.xml* file), which defines how URLs requested by a user are mapped to servlets. When analyzing the URL-to-servlet mapping, we take into account that not all servlets are directly accessible to users (those servlets that are not directly accessible are typically invoked internally by other servlets). Following the standard servlet invocation model, all URLs that point to accessible (public) servlets are assumed to be possible entry points.

Once the application's entry points are determined, the *Application Controller* systematically explores the state space of the application. To this end, it initiates execution of servlets by simulating all possible user choices of URLs. For example, if the application has three servlets mapped to the URLs */login*, */cart*, and */checkout*, the application controller attempts to execute all possible combinations (sequences) of these servlets. The actual order in which servlets are explored depends on the chosen *search strategy*. JPF offers a limited depth-first search (DFS) and a heuristics-based breadth-first search (BFS) strategy. We found that DFS works better for our system because it requires significantly less memory during model checking. With DFS, a path is explored until the system reaches a specific (configurable) limit on the number of entry points that are executed.

#### 4.2.3 State Space Management

Similar to other model checkers, Waler faces the state explosion problem. Thus, to make Waler scale to real-world web applications, we had to take a number of steps to manage (limit) the exponential growth of the application's state space. In particular, after careful analysis of several servlet-based applications, we found that JPF often fails to identify equivalent states. The two main reasons for that are: (1) the constraints added to the symbolic PC are never removed from it due to the design of JPF-SE[1], and (2), without domain-specific knowledge, JPF is not able to identify "logically equivalent" states. Here we present three techniques that we implemented to overcome these problems.

**States in JPF.** JPF comes with some mechanisms to identify equivalent states. A state in JPF is a snapshot of the current execution status of a thread, and it consists of the content of the stack, heap, and static variables storage. This snapshot is created when a sequence of executed instructions reaches a *choice* point, i.e., a point where there is more than one way to proceed from the current instruction. Choice points are thread-scheduling instructions, branching instructions that operate on symbolic values, or instructions where a new application entry point needs to be chosen. Whenever JPF finds a choice point, a snapshot of the current state is created. Then, the serialized version of the state is compared to hashes of previously-seen states. The execution path is terminated when the same state has been seen before.

We found that the basic version of JPF performs garbage collection and canonicalization of objects on the heap before hashing a state. However, it does not perform any additional analysis of memory content when comparing states for equality, as JPF has no knowledge of the domain-specific semantics of the objects in memory. As a result, JPF fails to recognize certain states as logically equivalent. This leads to a large number of states that are created unnecessarily. We discuss examples of some cases in which the standard JPF mechanism fails to identify equivalent states below.

**States in Waler.** In Waler, we extend the concept of JPF state to a "logical state" using the domain-specific knowledge that Waler has about web applications. In particular, we observe that the only information that is preserved between two user requests in a servlet-based application are the content of user sessions, application-level contexts, the symbolic PC (which stores constraints on symbolic variables stored in sessions), and data on persistent storage. Since we do not model persistent storage in Waler and always return a new symbolic value when it is accessed, we ignore this information in our analysis. Thus, the logical state of servlet-based application is defined as the content of user sessions and application contexts, and the PC. This is the only information that should be considered when comparing states after execution of a user request is finished.

**State space reduction.** Given the design of JPF and using our concept of logical state, we implemented three solutions to reduce the state space of a web application.

First of all, we implemented an additional analysis step to remove a constraint from the PC when it includes at least one variable that is no longer live[2]. This is especially important when the execution of a user request is finished, because, in a web application, input received by one servlet is independent from input received by another servlet, and, unless parts of it are stored in a persistent storage, any constraints on previous input are unrelated to the new one. The implemented solution is safe (it does not affect the soundness of the analysis) and allows our system to identify many states that are equivalent.

The second solution to reduce an application state space is to prune many "irrelevant" paths from state exploration. Consider, for example, an */error* servlet, which simply displays an error message, or a */products* servlet, which displays a list of available products. Executing such servlets often results in changes to the state of the memory, for example, due to different Java classes that must be loaded. However, once such a servlet is executed, the application is still in the same logical state. Also, the state after executing, for example, the servlet */login* will be logically equivalent to the state resulting from the execution of the sequence of servlets *[/error, /login]*. From this observation, it is clear that it would be beneficial to identify servlets whose executions do not modify the logical state of the application. The reason is that there is no need to consider them for vulnerability analysis. Therefore, after a servlet is executed, we analyze the content of the application's memory to determine whether the application logical state has been changed (for example, because of changes to the content of the user session). When no changes are detected, the exploration of the current execution path is terminated. This modification also does not compromise the soundness of the analysis, assuming that the memory analysis takes into the account all the component of the application logical state.

A third technique to limit the state space explosion problem is to identify irrelevant entry points, so that the servlets mapped to these URLs do not need to be executed. More precisely, during model checking, when our analysis determines that a servlet does neither read from nor write to the application's logical state at all, the execution of this page can be ignored for all other execution paths. The pruning of irrelevant servlets is especially helpful in large applications, where the execution of a servlet over symbolic inputs can take several minutes (and thus, can result in days of model checking time if the servlet is executed on multiple paths).

To summarize, the state explosion problem that can rise in the model checking of web applications can be significantly improved in many cases. In particular, we developed the following three techniques to limit the growth of an application's state space: we improved the existing JPF state hashing algorithm to disregard a path condition when its variables are out of scope, we found a way to prune the exploration of irrelevant paths, and we identify irrelevant servlets and discard them from our vulnerability analysis. We found that these techniques often allow for a significant reduction in the number of states explored by Waler. For example, running Waler on the *Jebbo-2* application (described in Section 5) without using any of our state reduction techniques resulted in the execution of 322,637 states, and it took around 223 minutes to terminate. When the same application was executed using our three heuristics, Waler terminated in about a minute and needed to explore only 529 states to obtain the same result.

```
1  public void _jspService(HttpServletRequest req,
2                          HttpServletResponse res) {
3
4    User user = (User) session.getAttribute("User");
5    if(user==null) {
6      User.adminLogin(request,response);
7      return;
8    }
9    ...
10   if(request.getMethod().equalsIgnoreCase("post")) {
11     result = website.variables.
12              insert(new Variable(req));
13   }
14 }
```

/admin/variables/Add.jsp

```
1  public void _jspService(HttpServletRequest req,
2                          HttpServletResponse res) {
3
4    User user = (User) session.getAttribute("User");
5    if(user==null || (!user.isAdmin())) {
6      User.adminLogin(request,response);
7      return;
8    }
9    ...
10   out.println("<a href=\"admin/variables/\
11            Add.jsp\">Add New</a>");
12 }
```

/admin/variables/index.jsp

Figure 3: Simplified version of an unauthorized access vulnerability in the JspCart application.

### 4.3 Vulnerability Detection

As described in the previous section, Waler uses model checking to systematically explore the state space of an application. During the model checking process, the system checks whether the likely invariants generated by Daikon for a program point hold whenever that point is reached. In our current implementation, we only consider likely invariants that are generated for exit points of methods (note that we differentiate between different exit points). The reason is that methods often check their parameters inside the function body (rather than in the caller). As a result, entry invariants are typically less significant.

To see an example of invariants that can be produced by our system, consider the code in Figure 3, which shows a vulnerability that Waler found in the JspCart applications (see Section 5). The left listing shows the code of the */admin/variables/Add.jsp* servlet, which is a privileged servlet that should only be invoked by an administrator. This is reflected by the set of likely invariants that are generated for the exit point on Line 14 for *Add.jsp*[3]:

```
(1) session.User != null
(2) session.User.isAdmin == true
(3) session.User.txtUsername == "admin@jspcart.com"
```

It can be seen that the first two invariants are part of the "true" program specification, while the third invariant

is spurious (an artifact of the limited test coverage). As a side note, the invariant for the exit point at *Add.jsp: Line 7* would be `session.User == null`.

To help us to determine whether a likely invariant holds or fails on a path, we implemented the *Program Checks Analyzer* module that keeps information about all the checks performed on an execution path. When a comparison instruction is executed, the PCA records the names of the variables involved and the result of the comparison. Also, the PCA keeps track of all variable assignments in the program. As a result, whenever the PCA encounters a check that operates on local variables, it can determine how this check constrains (affects) non-local variables. Recall that Daikon does not generate invariants for local variables, and, therefore, we are not interested in comparisons over local variables unless they store session data or method parameters.

Consider now what happens when Waler analyzes the *Add.jsp* servlet. After Waler executes the *if*-statement on Line 5, information about a new check is added to the set of current constraints accumulated by the PCA. If the user is authenticated, the value stored in the `session` object under the key `User` is not null. In this case, the PCA adds `session.User != null` to the set of checks along the current execution path, and the execution proceeds at Line 9[4]. Otherwise, the PCA records the fact `session.User == null`, and execution proceeds at Line 6.

Once the Line 14 of *Add.jsp* is reached, Waler checks whether all likely invariants generated for this point hold. A likely invariant holds on the current path if we can determine that the relationship among the involved variables is true. An invariant fails otherwise. To determine whether a likely invariant holds, we check whether the truth of this invariant can be determined directly given the current application state (i.e., the invariant involves concrete values). If not, we check whether the set of constraints accumulated on the current path implies the relationship defined by the invariant using the constraint solver employed by the JPF-SE.

Following the example, it can be seen that the first invariant for Line 14 always holds (because of the check on Line 5), while the other two might fail on some paths. In principle, we could immediately report the violations of the last two invariants as a potential program flaw. However, this would raise too many false positives, due to spurious invariants. In the following sections, we introduce two techniques to identify those invariants that are relevant to the detection of web application logic flaws.

#### 4.3.1 Supported Invariants

The first technique to identify real invariants is based on the insight that many vulnerabilities are due to developer oversights. That is, a developer introduces checks that enforce the correct behavior on most program paths, but misses an unexpected case where the correct behavior can be violated.

To capture this intuition, we defined a technique that keeps track of which paths contain checks that support an invariant and which paths are lacking such checks. More precisely, an execution path on which a likely invariant holds **and** it is supported by a set of checks on that path is added to the set of *supporting paths* for this invariant. That is, along a supporting path, the program contains checks that ensure that an invariant is true. A path on which a likely invariant can fail is added to the set of *violating paths*. When a likely invariant holds on all program paths to a given program point, then we know that it holds for all executions and there is no bug. When all paths can possibly violate a likely invariant, then we assume that the programmer did not intend this invariant to be part of the actual program specification, and it is likely an artifact of the limited test coverage. An application logic error is only reported by Waler if at least one *supporting path* and at least one *violating path* are found for an invariant at a program point.

Let us revisit the example of Figure 3. Waler determines that the first invariant on Line 14 of `Add.jsp` always holds. The third one is never supported, and, thus, it is correctly discarded as spurious. Moreover, Waler finds a violating path for the second invariant (`session.User.isAdmin == true`) by calling the *Add.jsp* servlet with a user in non-administrative role. However, the system also inspects the path where *index.jsp* is called first, which reflects the normal, intended flow of the application. This servlet, shown on the right of Figure 3, contains a check on Line 5 that adds the fact `session.User.isAdmin == true` to the PC (assuming that the user is authenticated as an administrator). In this case, when *Add.jsp* is invoked after *index.jsp*, the system determines that the invariant `session.User.isAdmin == true` holds and is supported. Thus, Waler finds a supporting path for this invariant. As a result, the fact that one can execute the main method of *Add.jsp* directly, violating its exit invariant `session.User.isAdmin == true`, is correctly recognized as an unauthorized access vulnerability.

We found that checking for supported invariants works well in practice. However, it can produce false positives and is not capable of capturing all possible logic flaws. The main source of false positives stems from the problem that the violation of an invariant, even when it is supported by a program check on some paths, does not necessarily result in a security vulnerability. For example, access to a normally protected page does not always result in a vulnerability because either (1) a sensitive operation performed by the page fails if a set of pre-



```
1 public void _jspService(HttpServletRequest req,
2                         HttpServletResponse res) {
3
4  if(req.getMethod() == "GET") {
5   ...
6   out.println("<form method=post"
7     + " action=\"edituser.jsp\">");
8   out.println("<input type=hidden"
9     + " name=\"username\" value="
10    + session.getAttribute("username") + ">");
11  ...
12  out.println("</form>");
13 }
14 if(req.getMethod() == "POST") {
15  ...
16  stmt = conn.prepareStatement("UPDATE users SET"
17    + " password = ?, name = ? WHERE username = ?");
18  stmt.setString(1, req.getParameter("password"));
19  stmt.setString(2, req.getParameter("name"));
20  stmt.setString(3, req.getParameter("username"));
21  stmt.executeUpdate();
22 }
23 }
```

edituser.jsp

Figure 4: Simplified user profile editing vulnerability (Jebbo-6).

```
1 public void doPost(HttpServletRequest req,
2                    HttpServletResponse res) {
3  ...
4  sess = request.getSession(true);
5  if(action.equals("/editpost")){
6    s = conn.prepareStatement("UPDATE posts SET"
7      + " author= ?, title = ?, entry = ?"
8      + " WHERE id = ?");
9    s.setString(1, (String)sess.getAttribute("auth"));
10   s.setString(2, req.getParameter("title"));
11   s.setString(3, req.getParameter("entry"));
12   s.setString(4, req.getParameter("id"));
13   s.executeUpdate();
14  }
15 }
```

PostController.java

Figure 5: Simplified post editing vulnerability (Jebbo-5).

conditions, uncontrolled by an attacker, is not satisfied, or (2) there is no sensitive operation on the path executed during the access. Reasoning about these cases is extremely hard for any automated tool. However, we found that such false positives often indicate non-security bugs in the code, and, thus, they are still useful for a developer. This technique also fails to identify logic vulnerabilities when the programmer does not introduce any checks for a security-relevant invariant at all. In such cases, Waler incorrectly concludes that an invariant is spurious because it cannot find any support in the code. To improve this limitation, we introduce an additional technique in the following section.

#### 4.3.2 Internal Consistency

As mentioned previously, Waler will discard invariants as spurious when they are not supported by at least one check along a program path. This can lead to missed vulnerabilities when the invariant is actually security-relevant. To address this problem, we leverage general domain knowledge about web applications and identify a class of invariants that we always consider significant, regardless of the presence of checks in the program.

We consider a likely invariant to be *significant* when it relates data stored in the user session with data that is used to query a database. Capturing this type of relationships is important because both the user session object and the database are the primary mechanism to store (persistent) information related to the logical state of the application. Moreover, we do not allow any arbitrary relationships: instead, we require that the invariant be an equality relationship. Such relationships are rarely coincidental because, by design, session objects and the database often replicate the same data.

Whenever Waler finds a path through the application that violates a significant invariant, it reports a logic vulnerability. To implement this technique, the system needed to be extended in two ways. First, we instrumented database queries so that the variables used in creating SQL queries are captured by Daikon and included into the invariant generation process. To this end, for each SQL query in the web application, we introduced a "dummy" function. The parameters of each function represent the variables used in the corresponding database query, and the function body is empty. The purpose of introducing this function is to force Daikon to consider the parameters for invariant generation at the function's exit point. Second, we require a mechanism to identify significant invariants. This was done in a straightforward fashion by inspecting equality invariants for the presence of variables that are related to the session object and database queries.

To see how the internal consistency technique can be used to identify a vulnerability, consider the code shown in Figure 4. This figure shows a snippet of code taken from the *edituser.jsp* servlet in one of the Jebbo applications (see Section 5)[5]. The purpose of this servlet is to allow users to edit and update their profiles. When the user invokes the servlet with a GET request, the application outputs a form, pre-filled with the user's current information. As part of this form, the application includes the user's name in the hidden field `username`, which is retrieved from the session object (shown in the upper half of Figure 4). When the user has finished updating her information, the form is submitted to the same servlet via a POST request. When this request is received, the application extracts the name of the user from the `username` parameter and performs a database query (lower half of Figure 4).

For this servlet, the dynamic analysis step (Daikon) generates the invariant `session.username == db_query.parameter3`, which expresses the fact

that a user can only update her own profile. Unfortunately, it is possible that a malicious client tampers with the hidden field `username` before submitting the form. In this case, the profile of an arbitrary user can be modified. Waler detects this vulnerability because it determines that there exists a path in the program where the aforementioned invariant is violated (as the parameter `username` is not checked by the code that handles the POST request). Since this invariant is considered significant, a logic flaw is reported.

The idea of checking the consistency of parameters to database queries can be further extended to also take into account the fields of the database that are affected by a query, but that do not appear explicitly in the query's parameters. Consider, for example, a message board application that allows users to update their own entries. It is possible that the corresponding database query uses only the identifier of the message entry to perform the update. However, when looking at the rows that are affected by legitimate updates, one can see that the name of the owner of a posting is always identical to the user who performs the update. To capture such consistency invariants, we extended the parameters of the "dummy" function to not only consider the inputs to the database query but to also include the values of all database fields that the query affects (before the query is executed). When multiple database rows are affected, the "dummy" function is invoked for each row, allowing Daikon to capture aggregated values of fields.

By extending the "dummy" function as outlined previously, Daikon can directly generate invariants that include fields stored in the database, even when these fields are not directly specified in the query parameters. Again, we consider invariants as significant if they introduce an equality relationship between database contents and session variables. The intuition is that these invariants imply a constraint on the database contents that can be accessed/modified by the query. If it was possible to violate such invariants, an attacker could modify records of the database that should not be affected by the query.

For example, this allows us to detect vulnerabilities where an attacker can modify the messages of other users in the Jebbo application. Consider the `doPost` function shown in Figure 5. The problem is that an authenticated user is able to edit the message of any other user by simply providing the application with a valid message `id`. During the dynamic analysis, the invariant `db.posts_author == session.auth` is generated, even though the `posts_author` field is not used as part of the update query. During model checking, we determine that this invariant can be violated (and report an alert) because there is no check on the `id` parameter that would enforce that only the messages written by the current user can be modified.

#### 4.3.3 Vulnerability Reporting

For each detected bug, Waler generates a vulnerability report. This report contains the likely invariant that was violated, the program point where this invariant belongs to, and the path on which the invariant was violated (given as a sequence of servlets and corresponding methods that were invoked). This information makes it quite easy for a developer or analyst to verify vulnerabilities. Currently, vulnerabilities are simply grouped by program points. Given the low number of false positives, this allows for an effective analysis of all reports. However, not every alert generated by Waler currently maps directly to a vulnerability or a false positive. We found several situations where several invariant violations referred to the same vulnerabilities (or a false positives) in application code. For example, Waler generated several alerts in situations when (conceptually) the same invariant is violated at different program points or when two distinct invariants refer to the same application's concept. Finding better techniques to aggregate and triage reports in such situations is an interesting topic of research, which we plan to investigate in the future.

#### 4.3.4 Limitations

Our approach aims at detecting logic vulnerabilities in a general, application-independent way. However, the current prototype version of Waler has a number of limitations, many of which, we believe, can be solved with more engineering. First, the types of vulnerabilities that can be identified by Waler are limited by the set of currently-implemented heuristics. For example, if an application allows the user to include a negative number of items in the shopping cart, we would be able to identify this issue only if the developer checked for that number to be non-negative on at least one program path leading to that program point. In addition, this check needs to be in a direct *if*-comparison[6] between variables. Conditions deriving from *switch* instructions or resulting from complex operations (such as regular expression matching) are not currently implemented.

Another limitation stems from the fact that we need a tool to derive approximations of program specifications. As a result, the detection rate of Waler is bounded by the capabilities of such a tool. In the current implementation, we chose to use Daikon. While Daikon is able to derive a wide variety of complex relationships between program variables, it has a limited support for some complex data structures. For example, if the `isAdmin` flag value is stored in a hash table, and it is not passed as an argument to any application function, Daikon will not be able to generate invariants based on that value. This limitation could be improved by implementing a smarter exploration technique for complex objects and/or by tracing local and temporary variables for the purpose of likely invariant generation. However, care needs to be exercised in this case to avoid an explosion in the number of invariants generated.

Another issue that we faced when working with Daikon was scalability: in its current implementation, Daikon creates a huge data structure in main memory when processing an execution trace. As a result, using Daikon on a larger application requires a large amount of RAM. We worked around this limitation by partitioning the application into subsets of classes and by performing the likely invariant generation on each subset separately.

A more import limitation of Daikon is that invariants generated by the tool cannot capture all possible relations. For example, the currently supported by Daikon invariants do not directly capture such temporal relations, as "operation A has to precede operation B." To address these limitations, different "intended behavior" capturing tools (such as [1]) could be employed by Waler in the first step of the analysis, although we leave this research direction for future work.

Another, more general, limitation of the first step of our analysis is the fact that we need to exercise the application in a "normal" way (i.e., not deviating from the developer's intended behavior). This part cannot be fully automated and needs human assistance. Nevertheless, many tools exist to ease the task of recording and scripting browsing user activity, such as Selenium [31].

Finally, the state explosion problem is one of the main limitations of the chosen model checking approach. We have already described several heuristics that help Waler limiting the state space of an application, and currently, we are working on implementing a combination of concrete and symbolic execution techniques to further improve scalability.

## 5 Evaluation

We evaluated the effectiveness of our system in detecting logic vulnerabilities on twelve applications: four real-world applications, (namely, *Easy JSP Forum*, *JspCart*[7], *GIMS* and *JaCoB*), which we download from the SourceForge repository [28], and eight servlet-based applications written by senior-level undergraduate students as part of a class project, named *Jebbo*. When choosing the applications, we were looking for the ones that could potentially contain interesting logic vulnerabilities, were small-enough to scale with the current prototype of Waler, and did not use any additional frameworks (such as Struts or Faces). While we show that it is possible to scale Waler to real-world applications, its scalability is still a work in progress as it is based on two tools, JPF and Daikon, that were not designed to work on large applications.

All chosen applications were analyzed following the techniques introduced in Section 4. During the model checking phase, we explored paths until a depth of 6 (that is, the limit for the depth-first search of JPF was set to 6). Note that all vulnerabilities reported below were found at depth of three or less; we then doubled the search depth to let Waler check for deeper bugs. All tests were performed on a PC with a Pentium 4 CPU (3.6 GHz) and 2 Gigabytes of RAM.

The results of our analysis are shown in Table 1. Waler found 29 previously-unknown vulnerabilities in four real-world applications and 18 previously-unknown vulnerabilities in eight Jebbo applications. It also produced a low number of false positives. In Table 1, the columns *Lines of Code* and *Bytecode Instructions* show the size of the applications in terms of the number of lines of Java code (JSP pages were first compiled into their servlet representations) and of the number of bytecode instructions, respectively. The column *Entry Points* shows how many entry points were found and analyzed by Waler and the column *States Explored* shows how many states were covered. The columns *Likely Invariants* and *Invariants Violated* respectively show how many invariants were generated by Daikon and how many of them were reported as violated by Waler. The numbers in the column *Alerts* represent the (manual) aggregation of the reported invariants violations (as it is discussed in Section 4.3.3). The columns *Vulnerabilities*, *Bugs*, and *False Positives* show the aggregated number of vulnerabilities, security-unrelated bugs, and false alarms that were produced by Waler. Note that the numbers on these columns are based on the analysis of the aggregated alerts. Finally, the column *Running Time* shows the time required for the analysis.

### 5.1 Vulnerabilities

**Easy JSP Forum:** The first application that we analyzed is the Easy JSP Forum application, a community forum written in JSP. Using Waler, we found that any authenticated user can edit or delete any post in a forum. To enforce access control, the Forum application does not show a "delete" or "edit link" for a post if the current user does not have moderator's privileges for the current forum but fails to check these privileges when a delete or an edit request is received. Thus, if a user forges a delete/edit request to the application using a valid post id (all ids can be obtained from the source code of web pages accessible by all users), a post will be deleted/modified.

**GIMS:** The second application that we analyzed is the Global Internship Management System (*GIMS*) web application, a human resource management software. Using Waler, we found that many of the pages in the ap-

| Application | Lines of Code | Bytecode Instructions | Entry Points | States Explored | Likely Invariants | Invariants Violated | Alerts | Vulnerabilities | Bugs | False Positives | Runtime (min) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Easy JSP Forum* | 2,416 | 7,348 | 2 | 251,657 | 5,824 | 6 | 3 | 2 | 0 | 1 | 319 |
| *GIMS* | 6,153 | 11,269 | 40 | 36,228 | 6,993 | 55 | 27 | 23 | 2 | 2 | 88 |
| *JaCoB* | 8,924 | 15,129 | 38 | 26,809 | 81,832 | 0 | 0 | 0 | 0 | 0 | 79 |
| *JspCart* | 21,294 | 45,765 | 86 | 1,152,661 | 34,286 | 5 | 5 | 5 | 0 | 0 | 4,576 |
| *Jebbo* | | | | | | | | | | | |
| 1 | 1,027 | 2,304 | 16 | 1,725 | 8,777 | 2 | 2 | 2 | 0 | 0 | 1.5 |
| 2 | 1,882 | 4,227 | 20 | 529 | 7,767 | 3 | 2 | 0 | 0 | 2 | 1 |
| 3 | 1,438 | 2,993 | 17 | 195 | 7,388 | 2 | 2 | 2 | 0 | 0 | 1 |
| 4 | 1,182 | 2,709 | 8 | 73 | 4,474 | 3 | 3 | 0 | 2 | 1 | 0.5 |
| 5 | 804 | 2,025 | 8 | 59 | 2,792 | 3 | 3 | 1 | 0 | 2 | 0.5 |
| 6 | 1,524 | 3,709 | 19 | 268 | 5,159 | 9 | 9 | 6 | 3 | 0 | 0.5 |
| 7 | 1,499 | 2,826 | 15 | 398 | 3,342 | 10 | 5 | 4 | 1 | 0 | 0.5 |
| 8 | 1,463 | 2,782 | 15 | 1,031 | 8,468 | 15 | 6 | 3 | 3 | 0 | 1.2 |

Table 1: Experimental results.

plication do not have sufficient protection from unauthorized access. In particular, our tool correctly identified 14 servlets that can be accessed by an unauthenticated user (a user that is not logged in at all). Most of these pages do contain a check that ensures that there is some user data in a session (which is only true for authenticated users). When a check fails, the application generates output that redirects the client's browser to a login page. Unfortunately, at this point, the application does *not* stop to process the request due to a missing return statement. Moreover, we found that certain pages in the *GIMS* application that should only be accessible to users with administrative privileges do not have checks to confirm the role of the current user. As a result, nine administrative pages were correctly reported as vulnerable.

**JaCoB:** The third application is *JaCoB*, a community message board application that supports posting and viewing of messages by registered users. For this program, our tool neither found any vulnerabilities nor did it generate any false alert. However, closer analysis of the application revealed two security flaws, which could not be identified with the techniques used by Waler. For example, when a user registers with the message board or logs in, she is expected to provide a username and a password. Unfortunately, when this information is processed by the application, the password is simply ignored. Also, in this application, a list of all its users and their private information is publicly available. These two problems represent serious security issues; however, they cannot be detected by Waler because the program specification that can be inferred from the application's behavior does not contain any discrepancies with respect to the application's code.

**JspCart:** The fourth test application is *JspCart*, which is a typical online store. Waler identified a number of pages in its administrative interface that can be accessed by unauthorized users. In *JspCart*, all pages available to an admin user are protected by checks that examine the *User* object in the session. More precisely, the application verifies that a user is authenticated and that the user has administrative privileges. However, Waler found that four out of 45 pages are missing the second check. Therefore, any user that has a regular account with the store can access administrative pages and add, modify, or delete settings (e.g., the processing charge for purchases). A simplified version of one of these vulnerabilities is shown in Figure 3. Waler also found a logic vulnerability that allows an authenticated user to edit the personal information of another user by submitting a valid email address of an existing user. This vulnerability is similar to the one shown in Figure 4.

**Jebbo:** We analyzed a set of eight Jebbo applications that were written by senior-level undergraduate students as a class project. Jebbo is a message board application that allows its users to open accounts, post public messages, and update their own messages and personal information. Some of the applications also implement a message rating functionality. For this project, all students were provided with a description of the application to implement along with a set of rules (including security constraints) that were expected to be enforced by the application.

After running Waler on this set of applications, we found that six out of eight applications contained one or more logic flaws. Examples of the vulnerabilities found by Waler include the fact that unauthenticated users can post a message to the board, and the lack of authorization checks when users rate an existing message (e.g., in order to avoid for a user to rate its own messages). Ironically, most of the student followed the provided specification carefully and were checking that access to certain pages is limited to authenticated users only; however, due to various mistakes, the enforcing checks were not always sufficient. For example, common problems that we found are missing return statements on an error path and a failure to foresee all possible paths available to a user to access a certain functionality.

Waler identified a number of application logic flaws that are associated with unauthorized data modification, such as the possibility to edit personal information or posts belonging to another user. Some of the examples of these vulnerabilities are shown in Figure 4 and Figure 5. These vulnerabilities are classic examples of inconsistent usage of data by the application. It is interesting to observe that even though the students were aware of possible parameter tampering vulnerabilities, and, in many cases, they were very careful about checking user input for validity, they often failed to apply this knowledge to cases where there were multiple paths to the same program point.

The results for the Jebbo application demonstrate that logic flaws are hard to avoid, even in simple web applications. Almost all applications in this set were found to be vulnerable despite the fact that the students were given a clear program specification and knew basic web security practices. Given the class level of the students who were enrolled in the class, it is reasonable to assume that their programming skills are not far off from those of entry-level programmers. This, together with the fact that the complexity of real-world applications is much higher than the complexity of the Jebbo application, can be seen as an indication of how wide-spread web application logic flaws are. Moreover, it can be argued that many real-world application are, at least partially, written by students who are widely employed year-round as interns.

### 5.2 Discussion

As it is shown in Table 1, Waler generated a low number of false positives. Careful analysis of the alerts which did not represent a vulnerability revealed that the majority of them represent true weaknesses in code. These alerts were classified as bugs. We found that these bugs were either potential vulnerabilities that turned out to be unexploitable in particular situations or were not interesting for exploitation. For example, an unauthenticated user might be able to access a certain page, but this access does not contain any sensitive information. We classified the rest of the alerts as false positives.

We also carefully analyzed the applications for false negatives. We found that Waler missed some security problems, like the ones in JaCoB, but we consider these vulnerabilities to be out of scope as they cannot be detected using our approach. We also identified several cases where Waler missed vulnerabilities that should be detectable using the described approach. The main reason for such false negatives is the incomplete modeling of all application features in the current version of Waler. For example, Waler only identifies program checks in the form of *if*-statements, but in real applications, checks can be implemented using, for instance, database queries and regular expressions. Precise modeling of such constructs is left for future work.

The other way to evaluate the false negatives rate of Waler would be to run it on an application that has some known logic vulnerabilities. Unfortunately, we found a very limited number of such applications to be available, and none of them met all of our current selection criteria for test applications.

## 6 Related Work

Our work is related to several areas of active research, such as deriving application specifications, using specifications for bug finding, and vulnerability analysis of web applications. However, due to the limited space available, in this section we will only highlight the research that, in our opinion, is most related.

First, our approach is related to a number of approaches that combine dynamically-generated invariants with static analysis. For example, Nimmer and Ernst explore how to integrate dynamic detection of program invariants and their static verification on a set of simple stand-alone applications using Daikon and the ESC/Java static checker [27]. The invariants that are verified by the static checker on all paths are determined to be the real invariants for an application, and the invariants that could not be statically verified are shown as warnings to the user. The main goal of this research is to show the feasibility of the proposed approach rather than to find bugs. Another work that explores benefits of combining Daikon-generated invariants with static analysis is the DSD-Crasher tool by Csallner and Smaragdakis [8]. The main goal of this system is to decrease the false positives rate of a static bug-finding tool for stand-alone Java applications. Dynamically-generated invariants are used by the CnC tool (also based on ESC/Java) as assumptions on methods arguments and return values to narrow the domain searched by the static analyzer. In Waler, in contrast to both approaches, we do not assume that the invariants generated by Daikon are correct, and we only consider them to be clues for vulnerability analysis. Introducing our two additional techniques to differentiate between real and spurious invariants allows us to avoid many of the false positives due to limitations of the dynamic analysis step.

Our work is also related to the research on using an application's code to infer application-specific properties that can be used for guided bug finding. To the best of our knowledge, one of the first techniques that uses inferred specifications to search for application-specific errors is the work by Engler et al. [10]. Their goal is similar to ours in the sense that both works are trying to identify violations of likely invariants in applications. The way it is achieved, though, is very different in the two approaches.

While we infer specifications from dynamic analysis and check for possible violations in the code via symbolic execution, Engler's work carries out all the steps via static analysis: a set of given *templates* is used to extract a set of "beliefs" from the code. Afterward, patterns contradicting these "beliefs" are identified in the code. While some of the templates may be useful for web applications, most of the bugs they try to identify are relative to kernel and memory-unsafe programming languages operations. Moreover, we believe that having an additional source of information (i.e., dynamic traces) for application invariants makes our system more robust.

There is also recent work that uses statistical analysis and program code to learn certain properties of the application, with the goal of searching for application-specific bugs. For example, Kremenek et al. propose a statistical approach, based on factor graphs, to automatically infer which program functions return or claim ownership of a resource [21]. The AutoISES tool applies the idea of using statically-inferred specifications to the detection of vulnerabilities in the implementations of access control mechanisms for OS-level code [34]. The differences between these approaches and ours are similar to the ones with the Engler's work. Both approaches use statistical analysis to find violations of properties that must hold for all program points, and they do not require reasoning about the values of variables.

Learning invariants through dynamic analysis has already found application security purposes, mostly in order to train an Intrusion Detection System. Baliga et al. [4] employ Daikon to extract invariants on kernel structures from periodic memory snapshot of a non-compromised running system. After the training phase, these learned invariants are used to detect the presence of kernel rootkits that may have altered vital kernel structures. A conceptually similar approach has also been applied by Bond et al. [6] to Java code through instrumentation of the Java Virtual Machine. An initial learning phase is employed to record the calling context and call history for security-sensitive functions. Afterwards, the collected information is used to identify function invocations with an anomalous context. An anomalous context or history is considered an indicator of an attempt to divert the intended flow of the application, possibly by the exploitation of a logic error in the code. In that case, an alert is triggered or the execution is aborted.

Although both the techniques proposed by Baliga and Bong share with ours an initial dynamic learning phase, how the information is leveraged differs. For example, unlike the two approaches above, we do not assume that the likely invariants generated by the first phase are real invariants, rather we simply use them as hints for further analysis. In addition, while in our second phase we try to identify logic errors in the code by means of static analysis, they instead try to detect attacks being performed on a live system. Such run-time detection imposes an overhead, which results in the requirement for dedicated hardware for [4] and a 2%-9% penalty in performance for [6]. The authors of the latter work, in particular, traded some coverage of the code (limiting to security-related functions) in order to retain acceptable performance. Even though they focused on logic errors, a direct comparison with their evaluation environment was not possible, because of the different targets of the analysis. More precisely, they looked for bugs in the Java libraries triggered by Java applets, rather than bugs in Java-based web applications.

Another direction of research deals with protection of web service components against malicious and/or compromised clients. Guha et al. [15] employ static analysis on JavaScript *client* code in order to extract an expected client behavior as seen by the server. The server is then protected by a proxy that filters possibly malicious clients which do not conform to the extracted behavior.

Finally, our work is related to a large corpus of work, such as [16, 5, 7, 17, 18, 22, 26, 30, 33, 36, 23, 29], in the area of vulnerability analysis of web applications. However, most of these research works focus on the detection of or the protection against input-validation attacks, which do not require any knowledge of application-specific rules.

Among the approaches cited above, Swaddler [7] and MiMoSA [5] are tools developed by our group that look for workflow violation attacks in PHP-based web applications, using a number of different techniques (including Daikon-generated invariants). However, Waler's approach is more general and is able to identify any kind of a policy violation that is either reflected by a check in the application or that violates a consistency constraint.

Our work is also related to the QED tool presented in [23]. QED uses concrete model checking (with a set of predefined concrete inputs) to identify taint-based vulnerabilities in servlet-based applications. The main similarity between the two tools is that they both use a set of heuristics to limit an application's state space during model checking. Heuristics used by QED, however, are more specific to the taint-propagation problem and require an additional analysis step.

## 7 Conclusions

In this paper, we have presented a novel approach to the identification of a class of application logic vulnerabilities, in the context of web applications. Our approach uses a composition of dynamic analysis and symbolic model checking to identify invariants that are a part of the "intended" program specification, but are not enforced on all paths in the code of a web application.



We implemented the proposed approaches in a tool, called Waler, that analyzes servlet-based web applications. We used Waler to identify a number of previously-unknown application logic vulnerabilities in several real-world applications and in a number of senior undergraduate projects.

To the best of our knowledge, Waler is the first tool that is able to automatically detect complex web application logic flaws without the need for a substantial human (annotation) effort or the use of *ad hoc*, manually-specified heuristics.

Future work will focus on extending the class of application logic vulnerabilities that we can identify. In addition, we plan to extend Waler to deal with a number of frameworks, such as Struts and Faces. This will require creating "symbolic" versions of the libraries included in these frameworks. This initial development effort will allow us to apply our tool to a much larger set of web applications, since most large-scale, servlet-based web applications rely on one of these popular frameworks, and the lack of their support in Waler was a serious limiting factor when choosing real-world applications for the evaluation described in this paper.

## 8 Acknowledgments

We want to thank David Evans, Vinod Ganapathy, Somesh Jha, and a number of anonymous reviewers who gave us very useful feedback on a previous version of this paper.

## References

[1] Ammons, G., Bodík, R., and Larus, J. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2002), ACM, pp. 4–16.

[2] Anand, S., Pasareanu, C., and Visser, W. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2007), Springer.

[3] Anley, C. Advanced SQL Injection in SQL Server Applications. Tech. rep., Next Generation Security Software, Ltd, 2002.

[4] Baliga, A., Ganapathy, V., and Iftode, L. Automatic Inference and Enforcement of Kernel Data Structure Invariants. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual* (2008), pp. 77–86.

[5] Balzarotti, D., Cova, M., Felmetsger, V., and Vigna, G. Multi-module Vulnerability Analysis of Web-based Applications. In *Proceedings of the ACM conference on Computer and Communications Security (CCS)* (2007), pp. 25–35.

[6] Bond, M., Srivastava, V., McKinley, K., and Shmatikov, V. Efficient, Context-Sensitive Detection of Semantic Attacks. Tech. Rep. TR-09-14, UT Austin Computer Sciences, 2009.

[7] Cova, M., Balzarotti, D., Felmetsger, V., and Vigna, G. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (2007), pp. 63–86.

[8] Csallner, C., Smaragdakis, Y., and Xie, T. Article 8 (37 pages)-DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. In *ACM Transactions on Software Engineering and Methodology (TOSEM)* (April 2008).

[9] The Daikon invariant detector. `http://groups.csail.mit.edu/pag/daikon/`.

[10] Engler, D., Chen, D., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 57–72.

[11] Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., and Xiao, C. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming 69*, 1–3 (Dec. 2007), 35–45.

[12] Fossi, M. Symantec Global Internet Security Threat Report. Tech. rep., Symantec, April 2009. Volume XIV.

[13] Foundation, T. A. S. Apache Tomcat. `http://tomcat.apache.org/`.

[14] Grossman, J. Seven Business Logic Flaws That Put Your Website at Risk. `http://www.whitehatsec.com/home/assets/WP_bizlogic092407.pdf`, September 2007.

[15] Guha, A., Krishnamurthi, S., and Jim, T. Using static analysis for Ajax intrusion detection. In *Proceedings of the 18th international conference on World wide web* (2009), ACM New York, NY, USA, pp. 561–570.

[16] Halfond, W., and Orso, A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the International Conference on Automated Software Engineering (ASE)* (November 2005), pp. 174–183.

[17] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D., and Kuo, S.-Y. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the International World Wide Web Conference (WWW)* (May 2004), pp. 40–52.

[18] Jovanovic, N., Kruegel, C., and Kirda, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2006).

[19] Java pathfinder. `http://javapathfinder.sourceforge.net/`.

[20] Klein, A. Cross Site Scripting Explained. Tech. rep., Sanctum Inc., June 2002.

[21] Kremenek, T., Twohey, P., Back, G., Ng, A., and Engler, D. From Uncertainty to Belief: Inferring the Specification Within. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)* (November 2006), pp. 161–176.

[22] Livshits, V., and Lam, M. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the USENIX Security Symposium* (August 2005), pp. 271–286.

[23] Martin, M., and Lam, M. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *Proceedings of the USENIX Security Symposium* (July 2008), pp. 31–43.

[24] Microsystems, S. Java Servlet Specification Version 2.4. `http://java.sun.com/products/servlet/reference/api/index.html`, 2003.

[25] MIDDLEWARE, O. W. O. S. ASM. http://asm.objectweb.org/.

[26] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., AND EVANS, D. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the International Information Security Conference (SEC)* (May 2005), pp. 372–382.

[27] NIMMER, J., AND ERNST, M. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification* (2001).

[28] OPEN SOURCE SOFTWARE. SourceForge. http://sourceforge.net.

[29] PALEARI, R., MARRONE, D., BRUSCHI, D., AND MONGA, M. On race vulnerabilities in web applications. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (July 2008).

[30] PIETRASZEK, T., AND BERGHE, C. V. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detections (RAID)* (2005), pp. 372–382.

[31] SELENIUM DEVELOPMENT TEAM. Selenium: Web Application Testing System. http://seleniumhq.org.

[32] SPETT, K. Blind SQL Injection. Tech. rep., SPI Dynamics, 2003.

[33] SU, Z., AND WASSERMANN, G. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the Annual Symposium on Principles of Programming Languages (POPL)* (2006), pp. 372–382.

[34] TAN, L., ZHANG, X., MA, X., XIONG, W., AND ZHOU, Y. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In *Proceedings of the USENIX Security Symposium* (July 2008), pp. 379–394.

[35] VISSER, W., HAVELUND, K., BRAT, G., PARK, S., AND LERDA, F. Model Checking Programs. *Automated Software Engineering Journal 10*, 2 (Apr. 2003).

[36] XIE, Y., AND AIKEN, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the USENIX Security Symposium* (August 2006).

## Notes

[1]As a consequence of that, JPF includes constraints that are no longer relevant to the current execution into the application's state, preventing it from detecting otherwise equivalent states.

[2]Note that by using the simple strategy of removing all constraints that reference no longer live variables, we might potentially lose some of the implied constraints in the PC. This can reduce the effectiveness of the reduction of the state space, but it does not interfere with the soundness of the analysis.

[3]The names of the variables are generated as explained in Section 4.1.

[4]When session data is accessed on a path, the PCA records that fact, along with the key that was used. This is done by storing the item *session.<key>* in an attribute of the memory location that holds the reference to the object. The information is then propagated by JPF with each bytecode instruction that accesses this memory location.

[5]A similar vulnerability was found by Waler in the JspCart application. We use Jebbo as a simpler example.

[6]Note that our tool works on Java bytecode rather than source code. Therefore, loop exit conditions are implicitly included, as they are implemented in terms of *IF* opcodes.

[7]The code for the *JspCart* application is located in the SourceForge repository under the name *B2B eCommerce Project*.



# Baaz: A System for Detecting Access Control Misconfigurations

*Tathagata Das*
*Microsoft Research India*
*tathadas@microsoft.com*

*Ranjita Bhagwan*
*Microsoft Research India*
*bhagwan@microsoft.com*

*Prasad Naldurg*
*Microsoft Research India*
*prasadn@microsoft.com*

## Abstract

Maintaining correct access control to shared resources such as file servers, wikis, and databases is an important part of enterprise network management. A combination of many factors, including high rates of churn in organizational roles, policy changes, and dynamic information-sharing scenarios, can trigger frequent updates to user permissions, leading to potential inconsistencies. With Baaz, we present a distributed system that monitors updates to access control metadata, analyzes this information to alert administrators about potential security and accessibility issues, and recommends suitable changes. Baaz detects misconfigurations that manifest as small inconsistencies in user permissions that are different from what their peers are entitled to, and prevents integrity and confidentiality vulnerabilities that could lead to insider attacks. In a deployment of our system on an organizational file server that stored confidential data, we found 10 high level security issues that impacted 1639 out of 105682 directories. These were promptly rectified.

## 1 Introduction

In present-day enterprise networks, shared resources such as file servers, web-based services such as wikis, and federated computing resources are becoming increasingly prevalent. Managing such shared resources requires not only timely availability of data, but also correct enforcement of enterprise security policies.

Ideally, all access should be managed through a perfectly engineered role-based access control (RBAC) system. Individuals in an organization should have well-defined and precise roles, and access control to all resources should be based purely on these roles. When a user changes her role, her access rights to all shared resources should automatically change according to the new role with immediate effect.

In reality though, several organizations use disjoint access control mechanisms which are not kept consistent. Often, access is granted to individual users rather than to appropriate roles. To make matters worse, administrators and resource owners manually provide and revoke access on an as-needed and sometimes ad-hoc basis. As access requirements and rights of individuals in the enterprise change over time, it is widely recognized [19, 12, 5] that maintaining consistent permissions to shared resources in compliance with organizational policy is a significant operational challenge.

Incorrect access permissions, or *access control misconfigurations*, can lead to both *security* and *accessibility* issues. Security misconfigurations arise when a user who should not have access to a certain resource according to organizational policy, does indeed have access. According to a recent report [12], 50 to 90% of the employees in 4 large financial organizations had permissions in excess to what was entitled to their organizational role, opening a window of opportunity for insider attacks that can lead to disclosure of confidential information for profit, data theft, or data integrity violations. The 2007 Price Waterhouse Cooper survey on the global state of information security found that 69% of database breaches were by insiders [24]. On the other hand, accessibility misconfigurations arise when a user who should legitimately have access to an object, does not. Such misconfigurations, in addition to being annoyances, impact user productivity.

Security and accessibility misconfigurations occur due to several reasons. One contributing factor is the high rate of churn in organizations, and in organizational roles among existing employees, which necessitate changes in access permissions. In the same report [12], it was estimated that in one business group of 3000 people, 1000 organizational changes were observed over a period of few months. Another factor is the dynamic nature of information sharing workflows, where employees work together across organizational groups on short-term collaborations. When permissions are granted to shared resources for such collaborations, they are rarely revoked. In longer time-scales, organizations also update their policies in response to changing protection needs. Very often, these policies are not explicitly written down and system administrators, who have an operational view of security, may not have a global view of organizational needs, and may not be able to make these changes in a timely manner.

To make matters worse, very often, no *complete* high-level manifests exist, which correctly assign access permissions for a resource according to organizational policy. Consequently, given the large numbers of shared resources, different access control mechanisms and enterprise churn, it is difficult for administrators to manually

manage access control.

To address these limitations of existing access control management systems, we present Baaz, a system that monitors access control metadata of various shared resources across an enterprise, finds security and accessibility misconfigurations using fast and efficient algorithms, and suggests suitable changes.

To our knowledge, Baaz is the first system that helps an administrator audit access control mechanisms and discover critical security and accessibility vulnerabilities in access control without using a high-level policy manifest. To do this, Baaz uses two novel algorithms: **Group Mapping**, which correlates two different access control or group membership datasets to find discrepancies, and **Object Clustering**, which uses statistical techniques to find slight differences in access control between users in the same dataset.

We do not claim that techniques we use in Baaz will find all misconfigurations, as the notion of policy itself is not defined in most of our deployment settings. Also, given that access permissions change very organically over time and several of these changes are linked to ad-hoc and one-off access requirements, it is very difficult for an automated system to deduce the exact and complete list of all misconfigurations. However, our deployment experiences with real datasets have shown Baaz to be very effective at flagging high-value security and accessibility misconfigurations.

The operational context and main characteristics of Baaz are:

- **No assumption of well-defined policy:** Baaz does not require a high-level policy manifest, though it can exploit one if it exists. Rather than checking for "correct" access control, it checks for "consistent" access control by comparing users' access permissions and memberships across different resources.
- **Proactive vs Reactive:** Baaz takes as input static permissions, such as access control lists, rather than access logs. This approach helps fix misconfigurations *before* they can be exploited, reducing chances of insider attacks. However, the system can be easily augmented to process access logs if required.
- **Timeliness:** Baaz continuously monitors access control, so it can be configured to detect and report misconfigurations on sensitive data items as they occur, or just present periodic reports for less sensitive data.

We present results from Baaz deployments on three heterogeneous resources across two organizations, We interacted with system administrators of both organizations to validate the reports and found a number of high-value security and accessibility misconfigurations, some of which were fixed immediately by the respective system administrators. In all these organizations, no policy manifest was readily available. Before we deployed Baaz, these administrators had to examine thousands of individual or group permissions to validate whether these permissions were intended. The utility of Baaz can be gauged to some extent from some comments we received from administrators:

> *"This report is very useful. I didn't even know these folks had access!"*
>
> *"This output tells me how many issues there are. Now I HAVE to figure out what to do in the future to handle access control better."*
>
> *"I did not realize that our policy change had not been implemented!"*

Our Baaz deployment in one organization found 10 security and 8 accessibility misconfigurations in confidential data stored on a shared file server. The security misconfigurations were providing 7 users unwarranted access to 1639 directories.

The rest of the paper is organized as follows: Section 2 describes our problem scope and assumptions. Section 3 presents the system architecture of Baaz, as well as an overview of our algorithm workflow. Section 4 explains our Matrix Reduction procedure for generating summary statements and reference groups, followed by Sections 5 and 6, in which we present our Group Mapping and Object Clustering algorithms. In Section 7, we outline more detailed issues we encountered while designing the system, and in Section 8, we describe our implementation, deployment and evaluation of the Baaz prototype. Related work is presented in Section 9, and Section 10 summarizes the paper.

## 2 System Assumptions

The main goal of Baaz is to find misconfigurations in access control permissions (as in ACLs) typically caused by inadvertent misconfigurations, which are difficult for an administrator to detect and rectify manually. We do not detect misconfigurations of access permissions caused by manipulation by active adversaries. We assume that the inputs to our tool, such as the ACLs and well-known user groups, are not tampered. In many organizations, only administrators or resource owners will be able to view and modify these metadata in the first place, so this assumption is reasonable.

In our target environment, a definition of correct policy is not explicitly available. Therefore, rather than checking for correct access control, which we believe is difficult, the system checks for consistent access control. Essentially, Baaz finds relatively *small inconsistencies* in user permissions by comparing different sets of access control lists, or by comparing user permissions within the same access control list. We assume that *large differences* in access control are not indicative of misconfigurations. Clearly, our definition of small inconsistencies and large differences (provided in Sections 5 and 6) will govern the set of misconfigurations we find. It is possible that this may lead to the system missing some genuine problems which is an inherent limitation. In fact, as described in Section 8.2, our deployment of Baaz missed detecting some valid misconfigurations. However, administrators can tune these parameters to keep the output concise and useful.

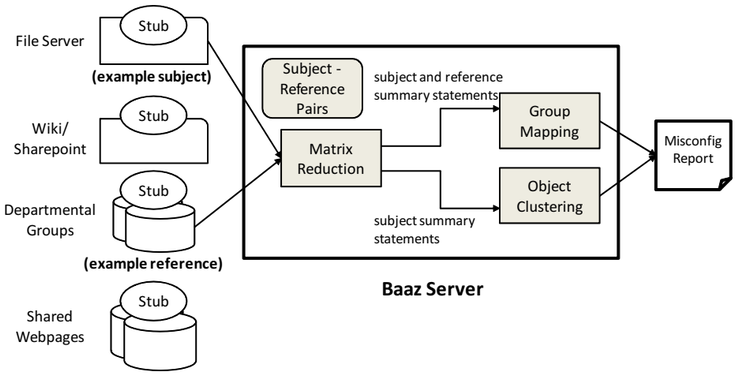



Figure 1: Baaz System Architecture

## 3 System Overview

In this section, we present an overview of the system components of Baaz. At the heart of our system, as shown in Figure 1, is a central server that collects access permission and membership change events from distributed *stubs* attached to shared resources. This server runs the misconfiguration detection algorithm when it receives these change events, and generates a report. An administrator/resource owner can decides whether each misconfiguration tuple that Baaz reports is valid, invalid, or an intentional exception. Administrators/owners will need to fix the valid misconfigurations manually. We now provide an overview of the client stubs and server functions.

### 3.1 Baaz Client Stubs

Baaz stubs continuously monitor access control permissions on shared resources such as file servers, wikis, version-control systems, and databases, and they monitor updates to memberships in departmental groups, email lists, etc. Each stub translates the access permissions for a shared resource into a binary **relation matrix**, an example of which is shown in Figure 2. Each such matrix captures relations specific to the resource that the stub runs on. For example, a file server stub captures the user-file access relationship, relating which users can access given files. On a database that stores organizational hierarchy, the Baaz stubs capture the user-group membership relation, relating which users are members of given groups. We shall refer to an element in the relation matrix $M$ as $M_{i,j}$. A "1" in the $i^{th}$ row and the $j^{th}$ column of $M$ indicates the relation holds between the entity at row $i$ with the entity at column $j$, e.g., user $i$ can read file $j$, or user $i$ belongs to group $j$, whereas a "0" indicates that the relation does not hold.

Each Baaz stub sends $M_{i,j}$ to the Baaz server either periodically, or in response to a change in the relationship. Section 7.2 further describes various issues that we need to consider while designing and implementing stubs.

### 3.2 Baaz Server

At initial setup, an administrator registers pairs of **subject datasets** and **reference datasets** with the server, which form inputs to the server's misconfiguration detection algorithm. The subject dataset is the access control dataset which an administrator wants to inspect for misconfigurations. A reference dataset is a separate access control or group membership dataset that Baaz treats as a baseline against which it compares the subject. In a sense, one can view the subject dataset as the implementation, and the reference dataset as an approximate policy, and the process of misconfiguration detection compares the implementation with the approximate policy.

Figure 2 shows an example subject dataset relation matrix of ten users (labeled as $A$ to $J$) and 16 objects (labeled as 1 to 16), and Figure 3 shows an example reference dataset relation matrix of the same set of users

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | | | | | | | | | 1 | 1 | 1 | 1 | 1 | | | |
| B | | | | | | | | | 1 | 1 | 1 | 1 | 1 | | | |
| C | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | 1 | 1 | 1 | 1 | 1 | | 1 | 1 |
| D | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | 1 | 1 | 1 | 1 | 1 | | 1 | 1 |
| E | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | |
| F | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | |
| G | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | |
| H | 1 | 1 | 1 | 1 | 1 | | | | | | | | | | | |
| I | | | | | | | | | | | | | 1 | | | |
| J | | | | | | | | | | | | | | | | |

Figure 2: Example subject dataset's relation matrix

| | W | X | Y | Z |
|---|---|---|---|---|
| A | 1 | | 1 | |
| B | 1 | | 1 | |
| C | 1 | 1 | 1 | 1 |
| D | | 1 | | 1 |
| E | | 1 | | |
| F | | 1 | | |
| G | | 1 | | |
| H | | 1 | | |
| I | | | | |
| J | | 1 | | |

Figure 3: Example reference dataset's relation matrix

and 4 groups (labeled as $W$ to $Z$). We will use these example inputs to illustrate our misconfiguration detection algorithm.

Administrators can register multiple subject-reference pairs with the server, and each pair is processed independently, with the server periodically generating one misconfiguration report for each. If any changes are detected in matrices corresponding to a registered subject-reference pair, the server runs the misconfiguration detection algorithm, which has three steps:

**Matrix Reduction:** In the first step, the server reduces the subject and reference datasets' relation matrices to *summary statements* that capture sets of users that have similar access permissions and group memberships. Each summary statement can be thought of as a high-level statement of policy intent, gleaned entirely from the low-level relation matrices. We explain this procedure in Section 4.

**Group Mapping:** In this step, our goal is to uncover access permissions in the subject dataset that seem inconsistent with patterns in the reference dataset. Consider an example where the subject is a file server, and a reference is a list of departmental groups, as shown in Figure 1. Say a directory hierarchy on the file server can be accessed by all members in the human resources department in an organization, and by only one member of the facilities department. This has a high likelihood of being a security misconfiguration. Section 5 explains this procedure.

**Object Clustering:** Finally, in the Object Clustering phase, Baaz finds potential inconsistencies in the subject dataset by comparing summary statements for the subject that are "similar", but not the same. The main idea is that a user whose access permissions differ only slightly from that of a larger set of users could potentially be a misconfiguration. For example, if 10 users in the subject dataset can access a given set of 100 files, but say an 11th user can access only 99 of these files, Baaz flags a candidate accessibility misconfiguration. We describe this in Section 6.

The system reports security candidates as *"A user set U MAY NOT need access to object set O"* . Accessibility candidates are of the form *"A user set U MAY need access to object set O"* At this point, the administrator will need to identify reported misconfiguration candidates as "valid", "invalid", or "intentional exceptions", which are defined as follows.

**Valid:** The misconfiguration candidate is correct, and the administrator needs to make the recommended changes.

**Invalid:** The misconfiguration candidate is incorrect, and the administrator should not make the recommended changes.

**Intentional Exception:** The administrator should not make the recommended changes, but the candidate provides useful information to the administrator.

The *intentional exception* category captures all reported misconfigurations that correspond to exceptions which appear out of the ordinary but are legitimate. Administrators found these exceptions to be useful as they help check compliance and may, over time, become valid misconfigurations. An example of an intentional exception is a user who has just changed roles. To help with the transition, he still has access to some documents related to his previous role. Hence while his access should not be revoked at the current time, it should probably be in the near future.

The server archives candidates marked as invalid, and does not explicitly display them in future reports. The reports will, however, display intentional exceptions. Section 7.1 describes more specific issues related to server design and evaluation.

One of the important properties of our algorithms is that the misconfiguration candidates converge to a steady state. That is, if we run our Group Mapping and Object Clustering algorithms repeatedly starting from a given raw configuration, and if we resolve our misconfigurations as suggested, we will eventually (and fairly quickly) reach a state where no new candidates appear. This guarantee is what we call *internal consistency*. We will illustrate this through our examples in Sections 4 and



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | | | | | | | | | 1 | 1 | 1 | 1 | 1 | | | |
| B | | | | | | | | | 1 | 1 | 1 | 1 | 1 | | | |
| C | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | 1 | 1 | 1 | 1 | 1 | | 1 | 1 |
| D | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | 1 | 1 | 1 | 1 | 1 | | 1 | 1 |
| E | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | |
| F | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | |
| G | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | |
| H | 1 | 1 | 1 | 1 | 1 | | | | | | | | | | | |
| I | | | | | | | | | | | | | 1 | | | |
| J | | | | | | | | | | | | | | | | |

Subject Dataset Summary Statements

1. $\{C, D\} \rightarrow \{15, 16\}$
2. $\{C, D, E, F, G\} \rightarrow \{6, 7\}$
3. $\{A, B, C, D\} \rightarrow \{9, 10, 11, 12\}$
4. $\{A, B, C, D, I\} \rightarrow \{13\}$
5. $\{C, D, E, F, G, H\} \rightarrow \{1, 2, 3, 4, 5\}$

Figure 4: The result of the matrix reduction step on our example subject dataset's matrix.

| | W | X | Y | Z |
|---|---|---|---|---|
| A | 1 | | 1 | |
| B | 1 | | 1 | |
| C | 1 | 1 | 1 | 1 |
| D | | 1 | | 1 |
| E | | 1 | | |
| F | | 1 | | |
| G | | 1 | | |
| H | | 1 | | |
| I | | | | |
| J | | 1 | | |

Reference Dataset Summary Statements

1. $\mathbf{G_1}$ : $\{C, D, E, F, G, H, J\} \rightarrow \{X\}$
2. $\mathbf{G_2}$ : $\{A, B, C\} \rightarrow \{W, Y\}$
3. $\mathbf{G_3}$ : $\{C, D\} \rightarrow \{Z\}$

Figure 5: The result of the matrix reduction step on our example reference dataset's matrix.

5. The detailed proof is available on our webpage [1]. In the next three sections, we describe the server algorithm in detail.

## 4 Matrix Reduction

We apply the matrix reduction procedure on the relation matrices of both the subject and reference datasets. The goal of this step, in the context of the subject dataset, is to find summary statements relating sets of users (*user-sets*) that can access the same sets of objects (*object-sets*). Given a relation matrix, different kinds of summaries can be generated. Role mining algorithms [22, 25, 18, 28, 10], for example, try to find minimal overlapping sets of users and objects that have common permissions. In contrast, we find user-sets that have access to disjoint object-sets, as required by our misconfiguration detection algorithms. For the reference dataset, we find group membership summaries in a similar manner.

### 4.1 Subject Dataset

Our algorithm takes the relation matrix for the subject dataset as input, and examines each column, grouping together all objects that have identical column vectors. Essentially, it groups all objects that are accessible to an identical set of users.

Figure 4 shows the summary statements that our Matrix Reduction algorithm finds for the example shown earlier in Figure 2. Each greyscale coloring within the matrix represents a distinct summary statement. The list of summary statements that our algorithm yields is also shown in the figure. The first statement arises from users $C$ and $D$ having identical access rights, since they both have access to objects 15 and 16, and to no other object. We therefore interpret this in the following way: Users $C$ and $D$ have *exclusive* access to objects 15 and 16, i.e. no other user has access to these objects.

The Baaz server finds all such summary statements to completely capture the matrix. Next, it explicitly filters out all summary statements that involve only one user since our algorithm only looks for misconfigurations involving objects that are shared between more than one user. Figure 6 presents this algorithm in detail.

**Complexity:** Since the algorithm simply involves one sweep through the subject's relation matrix, grouping together identical columns, it runs in $O(nm)$ time, where $n$ is the number of users in the matrix and $m$ is the number of objects.

**EXTRACT SUMMARY STATEMENTS**

***Input:*** $M$ {binary relation matrix of all users $\mathcal{U}$ and all objects $\mathcal{O}$}
***Output:*** $\mathcal{S}$ {set of summary statements $[U_k \rightarrow O_k]$}
***Uses:*** $H$ {hashtable, indexed by sets of users, stores sets of objects}

1: $\mathcal{S} = \phi, H = \phi$
2: **for all** $o \in \mathcal{O}$ **do**
3: $U = Get\ User\ Set(M, o)$ // gets the set of users who can access $o$
4: **if** $H.contains(U)$ **then**
5: $O_U = H.get(U)$
6: $H.put(U, O_U \bigcup \{o\})$
7: **else**
8: $H.put(U, \{o\})$
9: **end if**
10: **end for**
11: **for all** $U_k \in H.keys$ **do**
12: $O_k = H.get(U_k)$
13: $\mathcal{S} = \mathcal{S} \bigcup \{[U_k \rightarrow O_k]\}$
14: **end for**
15: **return** $\mathcal{S}$

Figure 6: Algorithm to extract summary statements given the users and the access control matrix

[1] http://research.microsoft.com/baaz

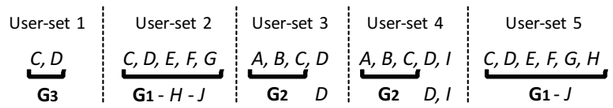



Figure 7: The result of the Group Mapping algorithm on the example subject matrix.

## 4.2 Reference Dataset

We apply the same process on the matrix for the reference dataset. The summary statements that our algorithm finds for the reference dataset relation matrix are shown in Figure 5. We call the user-set in each summary statement obtained from the reference dataset a *reference group*. The reference groups for our example are:

$\mathbf{G_1} = \{C, D, E, F, G, H, J\}$
$\mathbf{G_2} = \{A, B, C\}$
$\mathbf{G_3} = \{C, D\}$

The objects $W, X, Y, Z$ are merely used to find the reference groups, and are not used by future phases of our algorithm.

# 5 Group Mapping

In this section, we describe the Group Mapping algorithm, that takes as input the user-sets representing the subject dataset, and the reference groups discovered from the reference dataset, and finds the best mapping from the each user-set to the reference groups. The server uses these maps to flag outliers (users) as misconfiguration candidates. We first explain why Group Mapping is a useful step in finding misconfigurations. Next, we explain how Group Mapping works on our example data, and then we present the algorithm in detail.

## 5.1 Algorithm

Now we describe the Group Mapping algorithm in more detail. Table 1 summarizes the list of symbols and variables we use here, and in the description of the Object Clustering algorithm.

## 5.2 Intuition and Definitions

The Group Mapping algorithm for finding misconfigurations relies on the following two assumptions:

1. Users in the same reference group should have same access permissions.

2. Given a set of reference groups that have the same access permissions, any user who is not a member of these reference groups should not have the same access permissions as users within these reference groups.

Based on these two assumptions, we define misconfiguration candidates for the algorithm to find as follows:

**Accessibility (based on Assumption 1):** If a majority of the members of a reference group all have access to a set of objects, and a minority do not have access to the same set of objects, then we flag the users without access as accessibility misconfiguration candidates.

**Security (based on Assumption 2):** Of all users in a user-set, if a majority of them form one or more reference groups, and a minority of users do not form any reference groups, we flag the minority of users as security misconfiguration candidates.

Following these definitions, the first thing to do is to find a mapping from user-sets to reference groups. However, since we are looking for outliers, we do not restrict the algorithm to finding an exact and complete mapping. Our goal is to find the "best-effort" mapping from user-sets to reference groups. In this process, some users in the user-sets may not map to any reference group, or a user-set may map to a reference group that has some extraneous users, who are not part of the user-set.

To illustrate with our running example, our Group Mapping algorithm maps the five user-sets in the summary statements we found in Figure 4 to the reference groups found in the Section 4.2 as shown in Figure 7. For the user-set of summary statement 1, the mapping is exact. For the user-set for statement 2, the best map is $\mathbf{G_1}$, which covers all users but also includes users $H$ and $J$ who are not in the user-set. For the user-set in summary statement 4, the best map is $\mathbf{G_2}$, while users $D$ and $I$ remain unmapped.

From this mapping, using the assumptions and definitions stated above, we infer the following misconfiguration candidates:

1. *From summary statement 2, users H and J MAY need access to objects 6, 7.*

2. *From summary statement 3, user D MAY NOT need access to objects 9, 10, 11, and 12.*

3. *From summary statement 4, users D and I MAY NOT need access to object 13.*

4. *From summary statement 5, user J MAY need access to objects 1, 2, 3, 4, and 5.*



| Symbol | Definition |
|---|---|
| $n$ | number of users |
| $m$ | number of objects |
| $l$ | number of summary statements/user-sets from subject dataset |
| $g$ | number of reference groups from reference dataset |
| $U_i \rightarrow O_i$ | $i^{th}$ summary statement for subject, with $U_i$ being the user-set and $O_i$ being the object-set |
| $\mathbf{G_j}$ | $j^{th}$ reference group |
| $C_i$ | set of groups used to cover user-set $U_i$ |
| $T_i$ | list of uncovered users in user-set $U_i$ after covering it by $C_i$ |
| $\Delta\mathbf{G_j}$ | list of users in $\mathbf{G_j}$ but not in user-set $U_i$, where $\mathbf{G_j} \in C_i$ |

Table 1: Table summarizing all symbols used to explain Group Mapping and Object Clustering

The second and third are security misconfiguration candidates, while the first and fourth are accessibility misconfiguration candidates. User-set 1 does not generate a misconfiguration candidate because the mapping is exact.

Fixing these misconfigurations will improve the mapping from user-sets to reference groups in future runs of the algorithm. For example, if the administrator removes user $D$'s access to objects 9, 10, 11 and 12, the next time the algorithm runs, the summary statement 3 will reduce to $\{A, B, C\} \rightarrow \{9, 10, 11, 12\}$. Group mapping will exactly map the new user-set to $\mathbf{G_2}$, and hence the number of misconfiguration candidates will reduce. This is what we mean by our algorithm reaching an internally consistent state, as mentioned in Section 3.2.

Note that in flagging these candidates, we may have missed some misconfigurations. For example, it is certainly possible that users $C$ and $D$ (forming group $\mathbf{G_3}$) *should not* have access to objects 15 and 16. But given that there is no definition of correct policy, a complete and correct list of misconfigurations cannot be expected. However, Baaz does ensure that the permissions are consistent across user-sets and the reference groups they map to.

Baaz can use role mining algorithms in the Matrix Reduction step to find possibly a larger number of summary statements. However, our definitions of misconfiguration and our algorithms hinge on the property of object-sets being *disjoint*, without which the system may find conflicting misconfiguration candidates. For example, if summary statement 3 included object 15, i.e. $\{A, B, C, D\} \rightarrow \{9, 10, 11, 12, 15\}$, the object 15 would be common to the object-sets of summary statements 1 and 3. Then, from summary statement 3, Group Mapping would suggest that $D$ should not have access to object 15, but the exact Group Map for summary statement 1 indicates that $D$ should have access to object 15. Hence, while Baaz could use role mining algorithms, and leverage richer and larger numbers of user-sets, it would need to include more logic to resolve such conflicts. Instead, we go with the approach of using the simple Matrix Reduction algorithm that provides object-disjoint user-sets.

In spite of its procedural limitations, administrators and resource owners in various domains have found Baaz's techniques very useful in finding genuine high-value misconfigurations. We show this through our evaluation in Section 8,

Say the Matrix Reduction step from Section 4 outputs a total of $l$ summary statements and $g$ reference groups. The input to the Group Mapping step is the set of user-sets $U = \{U_1, U_2, \cdots, U_l\}$ from the summary statements, and the set of reference groups $\mathcal{G} = \{\mathbf{G_1}, \mathbf{G_2}, \cdots, \mathbf{G_g}\}$. Our objective can now be expressed in terms of finding a *set cover* for each user-set $U_i$ using a subset of the groups in $\mathcal{G}$. A set cover, in its usual sense, implies that the union of the covering subsets is exactly equal to the set to be covered. But, we are interested in finding an *approximate set cover*, where the cover need not be exhaustive, and reference groups could include members that are not in the user-set. The idea is to find a *maximal* overlap between the subject dataset user-sets and the reference groups. This *approximate set cover* $C_i$ may contain a group $\mathbf{G_j}$ such that some users in $\mathbf{G_j}$ are absent in $U_i$, as shown in Figure 7 with user-sets 2 and 5. Also, it is not necessary that $C_i$ covers *every* user in $U_i$, as shown with user-sets 3 and 4. We refer to the set of uncovered users in $U_i$ as $T_i$, i.e., is $T_i = U_i - \bigcup_{\forall \mathbf{G_j} \in C_i} \mathbf{G_j}$.

We choose an approximate set cover based on the *minimum description length (MDL)* principle [11], which ensures that the overlap is large, while the leftover set of uncovered users is small. In other words, $|C_i| + |T_i|$ is minimum over all possible *approximate set covers*. The *minimum* set cover problem is known to be NP-Hard, as it can take running time that is exponential on the set of users. By the same logic, the problem of finding *approximate set cover with minimum description length* is also NP-Hard. In practice, we have found that if the number of reference groups is less than 20, then it is feasible to solve it exactly on our testbed computers. For larger reference datasets, we use a well-known greedy approximation algorithm [16], which picks the set that has the maximal overlap, removes it from the reference set, and repeats the process. This is known to work within $O(log\ m)$ of optimal, where $m$ is the number of

**GROUP MAPPING**
***Input:*** $\mathcal{S}$ {summary statements}, $\mathcal{G}$ {reference groups}
***Output:***
$GAM$ {accessibility misconfigs [users,objects]},
$GSM$ {security misconfigs [users,objects]}

1: $GAM = \phi$ ; $GSM = \phi$
2: $U =$ all user-sets in the extracted summary statements $S$
3: **for all** $U_i \in U$ **do**
4: $(C_i, T_i) =$ Map Groups $(U_i, \mathcal{G})$
5: **for all** $\mathbf{G_j} \in C_i$ **do**
6: **if** $\frac{|\mathbf{G_j} - U_i|}{|U_i|} < 0.5$ **then**
7: $GAM = GAM \bigcup \{[\mathbf{G_j} - U_i, O_i]\}$
8: **end if**
9: **end for**
10: **if** $\frac{|T_i|}{|U_i|} < 0.5$ **then**
11: $GSM = GSM \bigcup \{[T_i, O_i]\}$
12: **end if**
13: **end for**
14: **return** $GAM, GSM$

**MAP GROUPS (APPROXIMATE)**
***Input:*** $U_i$ {set to be covered}, $\mathcal{G}$ {Groups}
***Output:*** $C_i$ {cover from $\mathcal{G}$}, $T_i$ {uncovered users in $U_i$}

1: $C_i = \phi$ ; $T_i = \phi$ ; $\mathcal{G}' = \phi$ ; $U_i' = U_i$
2: **for all** $G \in \mathcal{G}$ **do**
3: **if** $\frac{|\mathbf{G_j} - U_i|}{|U_i|} < 0.5$ **then**
4: $\mathcal{G}' = \mathcal{G}' \cup \{\mathbf{G}\}$
5: **end if**
6: **end for**
7: **repeat**
8: $MDL_{min} = MDL(U_i, C_i)$ ; $\mathbf{G_{min}} = \phi$
9: **for all** $\mathbf{G} \in \mathcal{G}'$ **do**
10: **if** $MDL(U_i, C_i \cup \{G\}) < MDL_{min}$ **then**
11: $\mathbf{G_{min}} = \mathbf{G}$
12: $MDL_{min} = MDL(U_i, C_i \cup \{\mathbf{G_{min}}\})$
13: **end if**
14: **end for**
15: **if** $\mathbf{G_{min}} = \phi$ **then**
16: **return** $C_i, U_i'$
17: **end if**
18: $C_i = C_i \bigcup \{\mathbf{G_{min}}\}$ ; $U_i' = U_i' - \mathbf{G_{min}}$
19: **until** $U_i' = \phi$
20: **return** $C_i, \phi$

Figure 8: Group Mapping Algorithm.

users in the user set, for the original minimum set cover problem. We modify this algorithm suitably to generate the *approximate set cover with minimum description length*.

Figure 8 shows the pseudocode for our Group Mapping algorithm. The main steps of the algorithm for a given list of user-sets $\{U_1, U_2, \cdots, U_l\}$ can be summarized as follows:

**Step 1:** For each user-set, first eliminate all groups in which more than half of the users are not members of the user-set (lines 2–6 in MAP GROUPS, Figure 8). Since less than half of the users in these reference group intersect with the user-set, this reference group will not figure in either security or accessibility misconfiguration candidates as defined in Section 5.2.

**Step 2:** When the number of groups in $\mathcal{G}$ is less than 20, we exhaustively search for all set covers and use the minimum. For larger $\mathcal{G}$, we use a modified version of the greedy set-cover algorithm to do the matching, as shown in Figure 8. For each user-set $U_i$, we pick a group $\mathbf{G}$ that overlaps maximally with $U_i$ (pick any one in case of ties). To apply the *minimum description length* principle, we define the description length for $U_i$ in terms of $\mathbf{G}$ as $|U_i - \mathbf{G}| + |\mathbf{G} - U_i|$. For example, in user-set 2, two potential mappings are $\mathbf{G_1}$ as shown in the example, or $\mathbf{G_3}$, which contains users $C$ and $D$. In the former case, $|U_2 - \mathbf{G_1}|$ is 0, and $|\mathbf{G_1} - U_2|$ is 2, since $\mathbf{G_1}$ contains two extraneous users, $H$ and $J$. In the latter mapping, $|U_2 - \mathbf{G_3}|$ is 3, since $\mathbf{G_3}$ covers $C$ and $D$ and does not include $E$, $F$ and $G$. Also, $|\mathbf{G_3} - U_2|$ is 0. Therefore the MDL metric for the former cover is 2, while in the latter case it is 3. Hence our algorithm picks $\mathbf{G_1}$ as the cover. Refer to lines 8–14 in MAP GROUPS, Figure 8.

Add this selected group to the cover $C_i$. Remove the covered users from $U_i$ to get $U_i'$ and repeat until all users are covered, and the ones that cannot be covered by any group are output as $T_i$. Refer to lines 15–19 in MAP GROUPS, Figure 8.

Using this mapping, we can find both security and accessibility misconfigurations for each user set $U_i$ extracted from the summary statements ($U_i \rightarrow O_i$), as shown in lines 4–14 GROUP MAPPING, Figure 8. The summary statement can be rewritten as:

$$\{\mathbf{G_1'} \cup \cdots \cup \mathbf{G_c'} \cup T_i\} \rightarrow O_i.$$

where $\mathbf{G_j'} = \mathbf{G_j} \cap U_i, \forall \mathbf{G_j} \in C_i$. Let $\Delta\mathbf{G_j}$ be the users in $\mathbf{G_j}$ who are not in $U_i$. Note that Step 1 ensures that $\frac{|\Delta\mathbf{G_j}|}{|\mathbf{G_j}|} < 0.5$, that is $\Delta\mathbf{G_j}$ is a minority in $\mathbf{G_j}$. Based on the intuition provided in the previous section, we infer that users in $\Delta\mathbf{G_j}$ (if any) may require access to the objects $O_i$. Hence, the intended access should be $\{\mathbf{G_1} \cup \cdots \cup \mathbf{G_c} \cup T_i\} \rightarrow O_i$ and for each $\mathbf{G_j} \in C_i$ such that corresponding $\Delta\mathbf{G_j} \neq \phi$, the system reports accessibility misconfiguration candidate as:

*users in $\Delta\mathbf{G_j}$ MAY need access to $O_i$*

Finding security misconfiguration candidates is a slightly different process. Again, for a given user-set $U_i$, the users in $T_i$ are those that do not match any of the reference groups but still have access to $O_i$. If these users form a minority of the users in the user-set $U_i$, that is $\frac{|T_i|}{|U_i|} < 0.5$ and $T_i \neq \phi$, then the system infers that the intended access should be $\{\mathbf{G_1} \cup \cdots \cup \mathbf{G_c}\} \rightarrow O_i$ and all users in $T_i$ are reported to be security misconfiguration candidates as:

*users in $T_i$ MAY NOT need access to $O_i$*

Note that while we use metrics based on simple majority and minority to detect misconfiguration candidates, our prototype implementation supports any threshold value between 0 and 1. A higher threshold may find more valid misconfigurations but may also increase the number of false alarms.

**Complexity:** The group mapping run time is bounded as $O(k^2 lg)$, where $k$ is the maximum number of users in a reference group, $g$ is the number of reference groups and $l$ is the number of summary statements.

## 5.3 Misconfiguration Prioritization

When Baaz presents the misconfiguration report to the administrator, it lists the candidates in a priority order. Prioritization of candidate misconfigurations is important because administrators may not have the time to validate all misconfiguration candidates that Baaz outputs, as in Dataset 2 in Section 8. In such cases, a ranking function helps them focus their attention on the high-value candidates.

The main intuition behind our ranking function is that when the mismatches between a user-set and its covering reference group is smaller, the possibility of the misconfiguration candidate being a valid issue is higher. The formula used for prioritization of both accessibility and security candidates capture this measure of difference in similarity between a user-set and its cover.

For accessibility misconfigurations, for a given $U_i$, the system computes a priority over each reference group $G_j$ in $C_i$, as:

$$\mathcal{P}(accessibility\ misconfig) = 1 - \frac{\sum_{j=1}^{c} |\Delta\mathbf{G_j}|}{|U_i|}$$

For security misconfiguration candidates, we use the fraction of potentially unauthorized users to prioritize as follows. The smaller the fraction of uncovered users, the higher the priority.

$$\mathcal{P}(security\ misconfig) = 1 - \frac{|T_i|}{|U_i|}$$

# 6 Object Clustering

Our second technique for finding misconfiguration candidates is Object Clustering. This procedure uses only the summary statements as input and is therefore particularly useful in the absence of suitable reference groups.

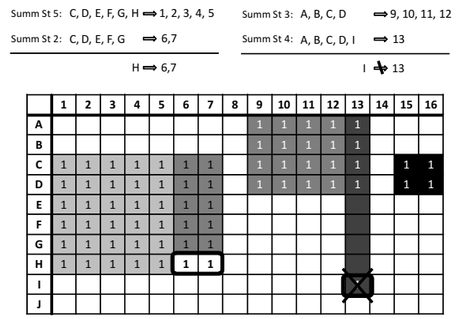



Figure 9: The result of the Object Clustering algorithm on the example subject matrix.

## 6.1 Intuition

We first present the intuition behind our Object Clustering algorithm. When the access permissions for a small user-set is only slightly different from the access control for a much larger user-set, this may indicate a misconfiguration.

Figure 9 explains this intuition using our example. Observe that the user-sets for summary statements 3 and 4 differ in one user – $I$ – because $I$ has access to object 13, but does not have access to any of 9, 10, 11 and 12. On the other hand, users $A$, $B$, $C$ and $D$ have access to objects 9, 10, 11, 12 and 13. Therefore, Baaz suggests a security misconfiguration candidate:

*user I MAY NOT need access to object 13.*

Similarly, summary statements 5 and 2 differ in only one user – $H$ – because $H$ does not have access to objects 6 and 7. Users $C$, $D$, $E$, $F$ and $G$ have access to 1, 2, 3, 4, 5, 6 and 7. Therefore, as shown in the figure, Baaz suggests an accessibility misconfiguration candidate:

*user H MAY need access to objects 6 and 7.*

The matrix in Figure 9 shows that if an administrator or resource owner determines that these are indeed valid misconfigurations and fixes them, the matrix becomes more uniform. A future iteration of matrix reduction will output fewer summary statements. In this example, $C$, $D$, $E$, $F$, $G$ and $H$ now have identical access and hence the reduction will remove summary statement 2. Similarly, since user $I$ will no longer have access to object 13, statement 4 will not be found in future iterations. This will lead to our algorithms finding the same number, or fewer misconfiguration candidates in the future, if no changes are made to the input matrices. This supports our claim of internal consistency in Section 3.2.

**OBJECT CLUSTERING**
***Input:*** $\mathcal{S}$ {summary statements}
***Output:*** $OAM$ {accessibility misconfigurations [users, objects]}, $OSM$ {security misconfigurations [users, objects]}

$OAM = \phi$ ; $OSM = \phi$
**for all** pairs of summary statements in $\mathcal{S}$, $[U1, O1]$ & $[U_2, O_2]$ **do**
  **if** $\frac{|U_1 - U_2|}{|U_1|} < 0.5$ and $\frac{|U_2 - U_1|}{|U_1|} < 0.5$ and $\frac{|O_2|}{|O_1|} < 0.5$ **then**
    **if** $U_1 - U_2 \neq \phi$ **then**
      $OAM = OAM \bigcup \{[U_1 - U_2, O_2]\}$
    **end if**
    **if** $U_2 - U_1 \neq \phi$ **then**
      $OSM = OSM \bigcup \{[U_2 - U_1, O_2]\}$
    **end if**
  **end if**
**end for**
**return** $OAM, OSM$

Figure 10: Object Clustering algorithm.

The Group Mapping and Object Clustering phases do not find disjoint sets of misconfigurations. For example, both the above misconfigurations were also flagged by Group Mapping. We intend to use Object Clustering as a fallback in situations where there do not exist suitable reference groups to flag misconfiguration candidates through Group Mapping.

## 6.2 Algorithm

We now describe the Object Clustering algorithm in detail. We first look for pairs of summary statements with the following template:

$$U_1 \rightarrow O_1 \textit{ and } U_2 \rightarrow O_2 \textit{ such that } \frac{|U_1 - U_2|}{|U_1|} < 0.5, \frac{|U_2 - U_1|}{|U_1|} < 0.5, \textit{ and } \frac{|O_2|}{|O_1|} < 0.5$$

Now, our definition of an object misconfiguration is as follows: For the two summary statements, $U_1 \rightarrow O_1$ and $U_2 \rightarrow O_2$ that match the template, say $|U_1 - U_2|/|U_1|$ and $|U_2 - U_1|/|U_1|$ are both smaller than 0.5 (a majority of users in $U_1$ are in $U_2$ and vice-versa), and $|O_2|/|O_1|$ is smaller than 0.5 ($O_2$ is less than half the size of $O_1$). We characterize a security misconfiguration candidate as:

$$U_2 - U_1 \textit{ MAY NOT need access to } O_2.$$

and an accessibility misconfiguration candidate is given as:

$$U_1 - U_2 \textit{ MAY need access to } O_2.$$

**Complexity:** Given that there are $l$ summary statements, $n$ users, and $m$ objects, the Object Clustering algorithm runs in $O(l^2(n+m))$ time.

## 6.3 Misconfiguration Prioritization

In the report, as in the case of Group Mapping, the Baaz server prioritizes these misconfigurations using the intuition that the more similar the user-sets $U_1$ and $U_2$, and the smaller the size of $O_2$, the higher the probability that the candidate is a genuine misconfiguration. The metric we use is the harmonic mean:

$$\mathcal{P}(misconfig) = 0.5 * \left((1 - \frac{|\Delta U|}{|U_1|}) + (1 - \frac{|O_2|}{|O_1|})\right)$$

Here $\Delta U$ corresponds to $U_2 - U_1$ or $U_1 - U_2$ depending on whether it is a security or an accessibility misconfiguration.

# 7 System Experiences

In this section, we describe issues that impact the quality of the misconfiguration reports produced by Baaz, based on our experiences in implementing and evaluating the Baaz server and stubs for our prototype, and discuss how we address them in our system design.

## 7.1 Server Design Issues

Here, we discuss our choice of reference dataset in our deployment and how an administrator can tune report time.

**Choosing reference datasets:** An administrator needs to use domain knowledge to choose the right reference dataset for a given subject dataset. We observe that the output reports vary depending on how rich or rigid the reference groups are. Some reference datasets, such as organizational group-membership relations, are rigid and structured, and contain few reference groups, potentially generating many misconfiguration candidates in the Group Mapping step, several of which may be invalid. This is because fewer groups will yield more approximate covers.

On the other hand, if a reference dataset contains a large number of reference groups, such as a set of email distribution lists, the report will contain fewer candidates because the chances of finding exact covers increases. As a result, the algorithm may not detect some valid misconfigurations. An administrator can decide which reference dataset to use, based on the sensitivity of the subject dataset, trading manual effort of validation for caution. For example, if a subject dataset folder is marked confidential, the administrator may choose to compare it with the organizational hierarchy, whereas email lists may be a better choice for less sensitive information.

In our evaluation described in Section 8, we choose email distribution lists as a reference dataset for two datasets and organizational hierarchy as a reference for one dataset, and our results verify our observations above.

**Tuning report time:** Since change events trigger Baaz's misconfiguration detection algorithms, the server may generate reports even in transient states while administrators manually change permissions. To avoid such spurious reports, each pair of subject and reference datasets has an associated *report time* ($T_r$): Baaz includes a candidate in its report only if it has existed for at least $T_r$ time. The administrator can configure $T_r$ to be short for subjects that store highly sensitive data, while it can be high for less important subjects. In our deployed prototype, we found that we could generate a report as fast as one second after a stub reports a change, or delay its reporting using $T_r$, as required.

## 7.2 Stub Design Issues

We identify two design issues that directly play a role in the quality of generated reports:

**Modeling access control:** The system's misconfiguration detection can only be as good as the data the stub provides. Access control mechanisms can be complicated [20], which sometimes makes capturing complete semantics in a stub quite hard. In our stub implementations, we have used a conservative approach towards modeling access control: if there is ambiguity of whether an individual or group has access to an object, we assume that they do indeed have access. This approach catches more security candidates albeit at the risk of increasing the number of false alarms. Previously proposed security monitoring systems have tackled this problem [6] using a similar strategy.

**Stub customization:** Access mechanisms of different kinds of resources will require custom stub implementations that can specifically understand the underlying access controls. Similarly, stubs may need to be customized to different data layouts containing group membership data. However, some stubs can be reused across resources. For example, in our prototype, we have implemented a stub that can run on any SMB-based Windows file share. We have also implemented customized stubs to capture organizational hierarchy and email lists within our enterprise, both of which reside on an Active Directory server [1] (an implementation of the Lightweight Directory Access Protocol, LDAP).

Access control permissions are not necessarily binary. For example, in a file share, "read-only" access or "full access" are only two of a number of different access types possible. Consequently, our stub implementations support various modes of operation. An administrator can choose what a "1" in the binary relation matrix captures: full access, read-only access, *any* kind of access, etc.

# 8 Evaluation

In this section, we first describe the implementation of Baaz system components (Section 8.1). Next, we describe the results we achieve through our prototype deployment (Section 8.2), followed by a description of the collection, analysis, and validation of misconfiguration reports from two other datasets (Section 8.3). Finally, we present performance evaluation microbenchmarks for demonstrating the scalability (Section 8.4) of the misconfiguration detection algorithms.

## 8.1 Implementation

We have implemented the Baaz server in C# using 2707 lines of code. We have also implemented Baaz stubs for an SMB-based Windows file server, for organizational groups in Active Directory [1], and for email distribution lists also stored in Active Directory. The Windows file server stub is entirely event-based: it traps changes in access control through the FileSystemWatcher [8] library and reports these changes immediately to the server. Currently, we only trap changes to access control for directories, but we can easily extend this to capture changes for individual files. The Active Directory stubs, on the other hand, poll the database every 8 minutes since we do not have the right permissions or mechanisms to build an event-based stub for Active Directory. The file server stub used 830 lines of C# code and the Active Directory stub, which used a common code base for both the organizational groups and email lists, was 1327 lines of C# code.

## 8.2 Evaluation Through Deployment

We have deployed Baaz within our organization, with stubs continuously monitoring two resources within our organization since August 19th, 2009. The stubs monitor read access permissions for directories on a Windows SMB file server that the employees use to share confidential data, and an Active Directory server storing email distribution lists relevant to the organization. Various groups within the organization actively use the file server to share documents, hence we found significant usage of access control capabilities on it.

The objective of our deployment was to see whether Baaz could help find valid access control misconfigurations on this file server. We therefore registered the file server as the subject dataset and the email distribution list as the reference dataset with the server. We decided to use email distribution lists as opposed to organizational hierarchy since our administrator observed that only organizational groups might not capture the various user sets that actively use the file server.

We show our results in three steps: first, we show how Baaz's first report in the deployment was effective in finding misconfigurations. Second, we show the utility of continuously monitoring changes in access control to find misconfigurations. Third, we compare our results with the ground-truth we established by manually inspecting directory permissions on the file server, to detect how many actual misconfigurations Baaz was able to flag.

**First-time report:** Row 1 in Table 2 provides details on this dataset, and row 1 in Table 3 gives the classifica-

| Dataset | Subject | Reference | Users | Objects | Ref Groups | Summ Stmts |
|---|---|---|---|---|---|---|
| 1 | File Server | Email Lists | 119 | 105682 | 237 | 39 |
| 2 | Shared Web Pages | Email Lists | 1794 | 1917 | 3385 | 307 |
| 3 | Email Lists | Org Grps | 115 | 243 | 11 | 205 |

Table 2: Datasets used to evaluate Baaz.

| Set | Security | | | | | | | | Accessibility | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Group Mapping | | | | Object Clustering | | | | Group Mapping | | | | Object Clustering | | | |
| | Tot. | Val. | Exc. | Inv. | Tot. | Val. | Exc. | Inv. | Tot. | Val. | Exc. | Inv. | Tot. | Val. | Exc. | Inv. |
| 1 | 11 | 10 | 0 | 1 | 11 | 7 | 1 | 3 | 8 | 8 | 0 | 0 | 9 | 0 | 0 | 9 |
| 2 | 7 | 3 | 0 | 4 | 0 | 0 | 0 | 0 | 9 | 4 | 1 | 4 | 0 | 0 | 0 | 0 |
| 3 | 18 | 6 | 5 | 7 | 0 | 0 | 0 | 0 | 33 | 6 | 0 | 27 | 0 | 0 | 0 | 0 |

Table 3: Misconfiguration analysis for each report generated by Baaz.

tion of the *first-time* report that Baaz generated using the relation matrices that the stubs sent to the Baaz server initially. The total number of users in the organization is 149, the number of objects (directories) in the subject data set's relation matrix is 105682, and the total number of reference groups (or unique distribution lists) is 237. The matrix reduction phase on the subject dataset produced 39 summary statements.

Baaz flagged a total of 39 misconfiguration candidates. To validate these, we involved the system administrator and the respective resource owners of the directories in question.

*Security:* Of the 11 security candidates that Baaz found through Group Mapping, 10 were valid security issues which the administrator considered important enough to fix immediately. Object Clustering found 7 of these 10 security misconfigurations, showing that Baaz would have been helpful in flagging security issues even if reference groups were not available to it. However it is clear that Group Mapping works more effectively than Object Clustering when a suitable reference dataset is available.

*Accessibility:* Baaz found 8 accessibility candidates through Group Mapping, all of which were valid. All 9 accessibility issues that Object Clustering flagged were invalid, showing that, with this dataset, while Group Mapping worked well in bringing out both security and accessibility issues, Object Clustering did well only with the security misconfigurations. Object Clustering was not effective in flagging valid accessibility issues since the difference between the summary statements were unexpectedly large.

Baaz found a total of 18 valid misconfigurations. There were 10 security misconfigurations involving 7 users which, when corrected, fixed access permissions on 1639 out of 105682 directories on the file server. There were 8 accessibility misconfigurations that affected 6 users and 163 directories.

Our deployment also helped us understand some of the reasons for why misconfigurations occur in access control lists, which we summarize below.

- In most cases, the misconfigurations arise because of employees changing their roles or, as in some accessibility issues, from new employees joining the organization.
- One of the security misconfigurations was caused by a policy change within the organization, which had only been partially implemented. Certain older employees had greater degree of access than newer employees since the administrator had inadvertently applied the policy change only to employees who had joined after the change was announced.
- A resource owner misspelt the name of one of the users they wanted to provide access to, inadvertently providing access to a completely unrelated employee.

**Real-time report:** In our deployment, the stubs and the server are running continuously, monitoring access control and group membership changes and subsequently running the misconfiguration detection algorithm. On September 20th, 2009, an employee within the organization adopted a new role, which was reflected by his addition to certain email distribution lists. The Baaz stub reported these changes to the server, following which the server reported one new accessibility misconfiguration candidate within one second. The administrator considered this accessibility misconfiguration important enough to rectify promptly. This emphasizes the value of Baaz's continuous monitoring approach since it enables administrators to detect misconfigurations in a nearly real-time fashion, just after they occur.

**Comparison to Ground-Truth:** To understand how close Baaz was to finding all misconfigurations for this file server, we manually examined access permissions of all directories on the file server from the root down to three levels. Beyond the third level, we only examined directories whose access permissions differed from their parent directories. We examined a total of 276 directories.

For each directory, we asked the directory owner two questions: If any user permissions to the directory should be revoked (security misconfiguration), and if anyone else should be provided access (accessibility misconfiguration). This procedure took two days to complete because of the manual effort involved. While we cannot claim that even this procedure would find all possible misconfigurations, we felt this exercise formed a good base-line to compare against Baaz.

We found that Baaz missed 4 security misconfigurations and 1 accessibility misconfiguration. Two security issues went undetected because an email list relevant to these issues was marked as private by the owner, and hence our Active Directory stub could not read the members. If we had the permission to run the stub with administrator privileges, Baaz would have flagged these issues. The other 3 issues (2 security and 1 accessibility) were genuinely missed by Baaz since there were no reference groups that matched the user-set, and the number of users involved in the misconfiguration (2) were more than half the size of the user-set (3).

Hence, while Baaz genuinely missed 3 misconfigurations, it did flag 18 valid misconfigurations which the administrator found very useful.

## 8.3 Snapshot Evaluation

We evaluated Baaz on two other subject and reference data pairs. We wrote stubs to gather snapshots of access control and group memberships from these datasets and generated a one-time report. Rows 2 and 3 of Table 2 describe the datasets and Table 3 summarize our findings. Dataset 2's subject is a server hosting shared internal web pages for projects and groups across an organization. The stub for this subject reads access permissions stored in an XML file in a custom format. The reference was, again, a set of email distribution lists created for this organization. This subject dataset comprised 1794 users and 1917 objects. For this dataset alone, the administrator decided to concentrate on misconfiguration candidates with priority more than 0.8.

In Dataset 3, the subject dataset is the set of email lists used as reference in Dataset 1, and the reference is the set of *organizational groups*. Here, each organizational group consists of a manager and all employees who report directly to the manager. As we have mentioned earlier, a reference dataset in Baaz may itself be inaccurate. Hence, this evaluation helps us check how stale the memberships to these email lists are. The number of users in this Dataset is 115 and the number of objects is 243. The slight discrepancy in the number of users in Datasets 1 and 3 is due to organizational churn in the period between when we ran the two experiments.

Baaz found many valid misconfigurations in all these datasets. Across all datasets, most security misconfigurations resulted due to role changes. Other security misconfigurations arose because an individual user, who had full permissions to an object, had inadvertently given access to another user who should not have had access. The causes of accessibility misconfigurations, similarly, were moves across organizations or inadvertent mistakes on the part of the individual manually assigning permissions.

We now summarize some other insights we acquired through this evaluation.

**Administrator input:** Baaz can only make recommendations. Only an administrator, or someone who has semantic knowledge about access requirements, needs to make the final decision of whether a misconfiguration is valid, an exception or invalid. For distributed access control systems such as Windows file servers, the validation will have to be through querying multiple people in the organization since objects involved in the misconfiguration can have different owners. This is not a simple task.

Despite this difficulty, overall, the administrators and resource owners found the system very useful since it found several valid security and accessibility misconfigurations. Moreover, what the administrators appreciated was that, instead of tracking down correct access for potentially thousands of objects, they needed to concentrate on a much smaller set of misconfiguration candidates that Baaz reports. For Datasets 1 and 3, the validation was mostly through conversation and email, and took approximately one hour. For Dataset 2, it took a total of three days turnaround time since we communicated only through email with resource owners who were at a remote site to complete the validation. Note that these are total turnaround times: it does not mean that an administrator spent three complete days just on the validation procedure.

**Group Mapping vs Object Clustering:** While Group Mapping is universally effective at finding misconfigurations, the Object Clustering approach is effective only in datasets which have a lot of statistical similarity. This is because Object Clustering relies on finding small deviations from a regular and often-repeated pattern of access control permissions. Datasets 2 and 3 do not have a regular pattern since most project web pages and email distribution lists had unique access permissions. Consequently, Object Clustering does not report any misconfigurations for these datasets. On the other hand, it does find misconfigurations for the file server (Dataset 1) since there were many directories on the file servers we evaluated with the same access permissions.

**Invalid Misconfigurations:** The number of invalid misconfigurations varies significantly across the different datasets. This is related to our discussion in Section 7.1.

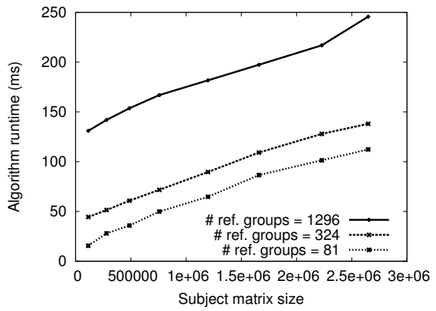



Figure 11: Scalability of the Baaz Algorithm

The organizational groups form a rigid reference dataset, so in Dataset 3, we see a large number of invalid misconfigurations. Across the datasets however, the number of invalid misconfigurations were small enough not to discourage an administrator in adopting our tool.

### 8.4 Algorithm Performance

In this Section, we concentrate on the performance and scalability of the server algorithm. We used Dataset 1 described in Table 2 for this experiment.

We ran the misconfiguration detection algorithm on the dataset while varying the subject relation matrix size, keeping the number of reference groups constant. To increase the matrix size, we increased the directory depth up to which we included objects into the subject's relation matrix, consequently increasing the number of objects, and therefore, the number of columns in the matrix.

Figure 11 shows the results of our experiments. Each line represents the algorithm's total run time which includes all three phases – Matrix Reduction, Group Mapping and Object Clustering – with different numbers of reference groups. We varied the number of reference groups by adding artificially created groups to the reference dataset while ensuring that the additional groups follow the same size distribution as the real reference groups. Every point in the graph is averaged across 20 runs. We ran all the experiments on a machine with a 3 GHz Intel Core 2 Duo CPU and 4 GB Memory, running a 64-bit version of Windows Server 2008.

With a matrix size of 2.7 million, and with 1296 reference groups, the misconfiguration detection takes a total of 246 ms to run. The increase in time is fairly linear in the matrix syst because the Matrix Reduction step dominates the total run-time of the algorithm. For the same data point, where Matrix Reduction needs to inspect roughly 2.7 million cells in the subject's relation matrix, Group Mapping needed to process only 24 summary statements and 1296 reference groups, and Object Clustering processed ${}^{24}C_2 = 276$ summary statement pairs.

Projecting from this graph, for a subject dataset representing 100,000 employees and 100,000 objects, i.e., a matrix size of $10^{10}$, and a reference dataset involving 1296 groups, the misconfiguration detection would take approximately 340 seconds to run. Our experiments indicate that the algorithm can scale to large datasets (much larger than encountered in our deployments as shown in Table 2), and run fast enough to provide prompt misconfiguration reports.

## 9 Related Work

In this section, we discuss our work in the context of related research.

Recent work by Baker et al. in detecting policy misconfigurations [4] uses data mining to infer association-rules between groups of resources that can be accessed by common sets of users, based on an off-line analysis of access attempts in log files. The authors use the profile and frequency of granted requests to predict and fix operational *accessibility* issues. For example, if a user belonging to such a common set inadvertently does not have access to a particular resource, their tool will flag this as a misconfiguration, and refer this to an appropriate resource owner.

Baaz on the other hand operates on access permissions. Consequently, in most cases, Baaz can flag and suggest fixes for misconfigurations before they can be exercised operationally. While access log analysis is an extremely useful mechanism in detecting security and accessibility issues, the approach is inherently complementary to the approach of analyzing access control permissions. Ideally, the two should be used in tandem.

Also, Baaz primarily uses a different technique, Group Mapping, whereby the system compares subject and reference datasets: several of the misconfigurations that the Group Mapping algorithm found in our evaluation could not have been found using association rules alone. These include the examples presented in Section 8.2 where users change roles, or new employees join an organization, and have not accessed any resources yet. In addition, Baaz finds both security and accessibility issues whereas Baker et al. concentrate only on accessibility issues.

Finally, the goal of their misconfiguration detection is similar in intent to Baaz's Object Clustering algorithm. While Baaz focuses on identifying sets of users that can access disjoint sets of objects, they identify all possible sets of users who have common access permissions to (possibly overlapping) sets of objects. In Baaz, we chose to focus on disjoint object-sets for reasons explained earlier.

Network intrusion prevention and detection systems also have a similar operational view of misconfigurations [15, 14]. An attempt is made to characterize normal behavior, as opposed to anomalous behavior, and any deviation from this characterization is flagged as a potential vulnerability. In contrast, research on automatically discovering attack graphs [2, 23], by correlating information across lists of known software-vulnerabilities, improper access controls, and network misconfiguration issues, have a forensic flavor. This aspect is further explored in more recent work such as HeatRay [6], which explores identity-snowball attacks based on over-entitled user privileges across a networked enterprise. The HeatRay tool outputs suggestions to administrators to prune privilege-lists on particular machines, maximizing security versus availability tradeoffs, using machine learning and combinatorial optimization techniques. A system such as Baaz can help an administrator decide whether to remove access permissions as suggested by HeatRay.

Other related work on policy anomaly detection includes the work on access control spaces [13] where the authors describe a policy-authoring tool called Gokyo that can help discover policy coverage issues.While Gokyo assumes a high-level policy manifest exists, Baaz works in scenarios where such manifests are not available.

Role-based access control (RBAC) [21] is widely cited as a useful management tool to control access permissions by separating out the user-role and role-permission relationships. However, RBAC is known to be difficult to implement in practice [5, 12]. The problem of role mining [22, 25, 18, 28, 10] is related to Baaz's matrix reduction step (Section 4), where we find related user and object groups. In role-mining, the user-object access matrix is analyzed to find maximal overlapping groupings of users and objects that have the same permissions. In contrast, in Baaz, we are interested in misconfigurations on shared-object permissions, as opposed to discovering common patterns of access across user groups. Nevertheless, like organizational groups, email groups, and distribution lists, the output of a role-mining algorithm, specifically the user-role mappings, can be used as an input to our group mapping phase. We believe that even if organizations adopt some flavor of RBAC, a system like Baaz is useful in discovering misconfigurations caused by exceptions and role changes. There is also a wealth of related work on the topic of clustering in general, and a summary of this is outside the scope of this work.

Policy anomaly detection is also a popular subject of study in the firewall and network configuration space. Here, existing tools [27] explore the semantics of different filtering rules and firewall policies. Testing and static analysis techniques [26, 17, 3] have been proposed to explore and understand how policy configurations satisfy properties such as redundancy and contradiction. However, all of these techniques are specific to firewall configurations and are inherently different from Baaz which uses comparison across ACL datasets and within the same dataset to find misconfigurations.

Several network security scanning tools are actively used by network administrators to find vulnerabilities such as open ports, vulnerable applications and poor passwords [7, 9]. Baaz's purpose and techniques target a different problem – finding access control misconfigurations – and are therefore complementary to the intent of these tools. In fact, a number of such tools and systems should be used in tandem to ensure a high level of security for all enterprise resources.

## 10 Conclusion

In this paper, we have described the design, implementation and evaluation of Baaz, a system used to detect access control misconfigurations in shared resources. Baaz continuously monitors access permissions and group memberships, and through the use of two techniques – Group Mapping and Object Clustering – finds candidate misconfigurations in the access permissions. Our evaluation shows that Baaz is very effective at finding real security and accessibility misconfigurations, which are useful to administrators.

## Acknowledgments

We would like to thank our shepherd, Somesh Jha, for his valuable comments and suggestions. We would also like to thank Ohil Manyam for testing and optimizing the prototype Baaz system, Rashmi K. Y, Geoffry Nordlund, and Chuck Needham for help with evaluating Baaz, and Geoffrey Voelker, Venkat Padmanabhan and Vishnu Navda for providing insightful comments that improved earlier drafts of this paper.

## References

[1] Active Directory. http://www.microsoft.com/windowsserver2003/technologies/directory/activedirectory/.

[2] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[3] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[4] L. Bauer, S. Garriss, and M. K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proc. SACMAT '08*, pages 185–194, New York, NY, USA, 2008. ACM.

[5] Bruce Schneier, Real-World Access Control. http://www.schneier.com/crypto-gram-0909.html.

[6] J. Dunagan, A. X. Zheng, and D. R. Simon. Heatray: Combating identity snowball attacks using machine learning, combinatorial optimization and attack graphs. *SIGOPS Oper. Syst. Rev.*, 2009.

[7] D. Farmer and E. H. Spafford. The COPS security checker system. In *Proceedings of the Summer Usenix Conference*, 1990.

[8] File System Watcher Class. http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.aspx.

[9] S. S. A. T. for Analyzing Networks. http://www.porcupine.org/satan.

[10] M. Frank, D. Basin, and J. M. Buchmann. A class of probabilistic models for role engineering. In *CCS '08*. ACM, 2008.

[11] P. D. Grunwald. *The Minimum Description Length Principle*. The MIT Press, 2007.

[12] Information Risk in the Professional Services-Field Study Results from Financial Institutions and a Roadmap for Research. http://mba.tuck.dartmouth.edu/digital/Research/ResearchProjects/DataFinancial.pdf.

[13] T. Jaeger, X. Zhang, and A. Edwards. Policy management using access control spaces. *ACM Trans. Inf. Syst. Secur.*, 6(3):327–364, 2003.

[14] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. *SIGOPS Oper. Syst. Rev.*, 39(5):91–104, 2005.

[15] S. T. King and P. M. Chen. Backtracking intrusions. *SIGOPS Oper. Syst. Rev.*, 37(5):223–236, 2003.

[16] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, 1994.

[17] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 177, Washington, DC, USA, 2000. IEEE Computer Society.

[18] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo. Mining roles with semantic meanings. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, 2008.

[19] Privileged Password Management: combating the insider threat and meeting compliance regulations for the enterprise. http://www.cyber-ark.com/constants/white-papers.asp?dload=IDC_White_Paper.pdf.

[20] M. Russinovich, D. Solomon, and A. Ionescu. *Windows Internals, 5th Edition*. Microsoft Press, 2009.

[21] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[22] J. Schlegelmilch and U. Steffens. Role mining with orca. In *Proc. SACMAT '05*, pages 168–176, 2005.

[23] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[24] The insider threat: automated identity and access controls can help organizations mitigate risks to important data. http://findarticles.com/p/articles/mi_m4153/is_2_65/ai_n25449309.

[25] J. Vaidya, V. Atluri, and J. Warner. Roleminer: mining roles using subset enumeration. In *CCS '06*, pages 144–153. ACM, 2006.

[26] A. Wool. Architecting the lumeta firewall analyzer. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 7–7, Berkeley, CA, USA, 2001. USENIX Association.

[27] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006.

[28] D. Zhang, K. Ramamohanarao, and T. Ebringer. Role engineering using graph optimisation. In *SACMAT '07*, pages 139–144. ACM, 2007.



# Cling: A Memory Allocator to Mitigate Dangling Pointers

*Periklis Akritidis*

*Niometrics, Singapore, and*
*University of Cambridge, UK*

## Abstract

Use-after-free vulnerabilities exploiting so-called dangling pointers to deallocated objects are just as dangerous as buffer overflows: they may enable arbitrary code execution. Unfortunately, state-of-the-art defenses against use-after-free vulnerabilities require compiler support, pervasive source code modifications, or incur high performance overheads. This paper presents and evaluates Cling, a memory allocator designed to thwart these attacks at runtime. Cling utilizes more address space, a plentiful resource on modern machines, to prevent type-unsafe address space reuse among objects of different types. It infers type information about allocated objects at runtime by inspecting the call stack of memory allocation routines. Cling disrupts a large class of attacks against use-after-free vulnerabilities, notably including those hijacking the C++ virtual function dispatch mechanism, with low CPU and physical memory overhead even for allocation intensive applications.

## 1 Introduction

Dangling pointers are pointers left pointing to deallocated memory after the object they used to point to has been freed. Attackers may use appropriately crafted inputs to manipulate programs containing use-after-free vulnerabilities [18] into accessing memory through dangling pointers. When accessing memory through a dangling pointer, the compromised program assumes it operates on an object of the type formerly occupying the memory, but will actually operate on whatever data happens to be occupying the memory at that time.

The potential security impact of these, so called, temporal memory safety violations is just as serious as that of the better known spatial memory safety violations, such as buffer overflows. In practice, however, use-after-free vulnerabilities were often dismissed as mere denial-of-service threats, because successful exploitation for arbitrary code execution requires sophisticated control over the layout of heap memory. In one well publicized case, flaw CVE-2005-4360 [17] in Microsoft IIS remained unpatched for almost two years after being discovered and classified as low-risk in December 2005.

Use-after-free vulnerabilities, however, are receiving increasing attention by security researchers and attackers alike. Researchers have been demonstrating exploitation techniques, such as heap spraying and heap feng shui [21, 1], that achieve the control over heap layout necessary for reliable attacks, and several use-after-free vulnerabilities have been recently discovered and fixed by security researchers and software vendors. By now far from a theoretical risk, use-after-free vulnerabilities have been used against Microsoft IE in the wild, such as CVE-2008-4844, and more recently CVE-2010-0249 in the well publicized attack on Google's corporate network.

Such attacks exploiting use-after-free vulnerabilities may become more widespread. Dangling pointers likely abound in programs using manual memory management, because consistent manual memory management across large programs is notoriously error prone. Some dangling pointer bugs cause crashes and can be discovered during early testing, but others may go unnoticed because the dangling pointer is either not created or not dereferenced in typical execution scenarios, or it is dereferenced before the pointed-to memory has been reused for other objects. Nevertheless, attackers can still trigger unsafe dangling pointer dereferences by using appropriate inputs to cause a particular sequence of allocation and deallocation requests.

Unlike omitted bounds checks that in many cases are easy to spot through local code inspection, use-after-free bugs are hard to find through code review, because they require reasoning about the state of memory accessed by a pointer. This state depends on previously executed code, potentially in a different network request. For the same reasons, use-after-free bugs are also hard to find through automated code analysis. Moreover, the combi-

nation of manual memory management and object oriented programming in C++ provides fertile ground for attacks, because, as we will explain in Section 2.1, the virtual function dispatch mechanism is an ideal target for dangling pointer attacks.

While other memory management related security problems, including invalid frees, double frees, and heap metadata overwrites, have been addressed efficiently and transparently to the programmer in state-of-the-art memory allocators, existing defenses against use-after-free vulnerabilities incur high overheads or require compiler support and pervasive source code modifications.

In this paper we describe and evaluate Cling, a memory allocator designed to harden programs against use-after-free vulnerabilities transparently and with low overhead. Cling constrains memory allocation to allow address space reuse only among objects of the same type. Allocation requests are inferred to be for objects of the same type by inspecting the allocation routine's call stack under the assumption that an allocation site (*i.e.* a call site of `malloc` or `new`) allocates objects of a single type or arrays of objects of a single type. Simple wrapper functions around memory allocation routines (for example, the typical `my_malloc` or `safe_malloc` wrappers checking the return value of `malloc` for `NULL`) can be detected at runtime and unwound to recover a meaningful allocation site. Constraining memory allocation this way thwarts most dangling pointer attacks —importantly— including those attacking the C++ virtual function dispatch mechanism, and has low CPU and memory overhead even for allocation intensive applications.

These benefits are achieved at the cost of using additional address space. Fortunately, sufficient amounts of address space are available in modern 64-bit machines, and Cling does not leak address space over time, because the number of memory allocation sites in a program is constant. Moreover, for machines with limited address space, a mechanism to recover address space is sketched in Section 3.6. Although we did not encounter a case where the address space of 32-bit machines was insufficient in practice, the margins are clearly narrow, and some applications are bound to exceed them. In the rest of this paper we assume a 64-bit address space—a reasonable requirement given the current state of technology.

The rest of the paper is organized as follows. Section 2 describes the mechanics of dangling pointer attacks and how type-safe memory reuse defeats the majority of attacks. Section 3 describes the design and implementation of Cling, our memory allocator that enforces type-safe address space reuse at runtime. Section 4 evaluates the performance of Cling on CPU bound benchmarks with many allocation requests, as well as the Firefox web browser (web browsers have been the main target of use-after-free attacks so far). Finally, we survey related work in Section 5 and conclude in Section 6.

# 2 Background

## 2.1 Dangling Pointer Attacks

Use-after-free errors are, so called, temporal memory safety violations, accessing memory that is no longer valid. They are duals of the better known spatial memory safety violations, such as buffer overflows, that access memory outside prescribed bounds. Temporal memory safety violations are just as dangerous as spatial memory safety violations. Both can be used to corrupt memory with unintended memory writes, or leak secrets through unintended memory reads.

When a program accesses memory through a dangling pointer during an attack, it may access the contents of some other object that happens to occupy the memory at the time. This new object may even contain data legitimately controlled by the attacker, *e.g.* content from a malicious web page. The attacker can exploit this to hijack critical fields in the old object by forcing the program to read attacker supplied values through the dangling pointer instead.

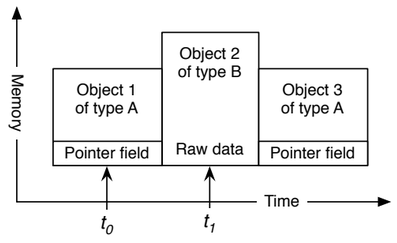



Figure 1: Unsafe memory reuse with dangling pointer.

For example, if a pointer that used to point to an object with a function pointer field (*e.g.* object 1 at time $t_0$ in Figure 1) is dereferenced to access the function pointer after the object has been freed, the value read for the function pointer will be whatever value happens to occupy the object's memory at the moment (*e.g.* raw data from object 2 at time $t_1$ in Figure 1). One way to exploit this is for the attacker to arrange his data to end up in the memory previously occupied by the object pointed by the dangling pointer and supply an appropriate value within his data to be read in place of the function pointer. By triggering the program to dereference the dangling pointer, the attacker data will be interpreted as a function pointer, diverting program control flow to the location dictated by the attacker, *e.g.* to shellcode (attacker code smuggled into the process as data).

Placing a buffer with attacker supplied data to the exact location pointed by a danging pointer is complicated by unpredictability in heap memory allocation. However, the technique of heap spraying can address this challenge with high probability of success by allocating large amounts of heap memory in the hope that some of it will end up at the right memory location. Alternatively, the attacker may let the program dereference a random function pointer, and similarly to uninitialized memory access exploits, use heap spraying to fill large amounts of memory with shellcode, hoping that the random location where control flow will land will be occupied by attacker code.

Attacks are not limited to hijacking function pointers fields in heap objects. Unfortunately, object oriented programming with manual memory management is inviting use-after-free attacks: C++ objects contain pointers to virtual tables (`vtables`) used for resolving virtual functions. In turn, these `vtables` contain pointers to virtual functions of the object's class. Attackers can hijack the `vtable` pointers diverting virtual function calls made through dangling pointers to a bogus `vtable`, and execute attacker code. Such `vtable` pointers abound in the heap memory of C++ programs.

Attackers may have to overcome an obstacle: the `vtable` pointer in a freed object is often aligned with the `vtable` pointer in the new object occupying the freed object's memory. This situation is likely, because the `vtable` pointer typically occupies the first word of an object's memory, and hence will be likely aligned with the `vtable` pointer of a new object allocated in its place right after the original object was freed. The attack is disrupted because the attacker lacks sufficient control over the new object's `vtable` pointer value that is maintained by the language runtime, and always points to a genuine, even if belonging to the wrong type, `vtable`, rather than arbitrary, attacker-controlled data. Attackers may overcome this problem by exploiting objects using multiple inheritance that have multiple `vtable` pointers located at various offsets, or objects derived from a base class with no virtual functions that do not have `vtable` pointers at offset zero, or by manipulating the heap to achieve an exploitable alignment through an appropriate sequence of allocations and deallocations. We will see that our defense prevents attackers from achieving such exploitable alignments.

Attacks are not limited to subverting control flow; they can also hijack data fields [7]. Hijacked data pointers, for instance, can be exploited to overwrite other targets, including function pointers, indirectly: if a program writes through a data pointer field of a deallocated object, an attacker controlling the memory contents of the deallocated object can divert the write to an arbitrary memory location. Other potential attacks include information leaks through reading the contents of a dangling pointer now pointing to sensitive information, and privilege escalation by hijacking data fields holding credentials.

Under certain memory allocator designs, dangling pointer bugs can be exploited without memory having to be reused by another object. Memory allocator metadata stored in free memory, such as pointers chaining free memory chunks into free lists, can play the role of the other object. When the deallocated object is referenced through a dangling pointer, the heap metadata occupying its memory will be interpreted as its fields. For example, a free list pointer may point to a chunk of free memory that contains leftover attacker data, such as a bogus `vtable`. Calling a virtual function through the dangling pointer would divert control to an arbitrary location of the attacker's choice. We must consider such attacks when designing a memory allocator to mitigate use-after-free vulnerabilities.

Finally, in all the above scenarios, attackers exploit reads through dangling pointers, but writes through a dangling pointer could also be exploited, by corrupting the object, or allocator metadata, now occupying the freed object's memory.

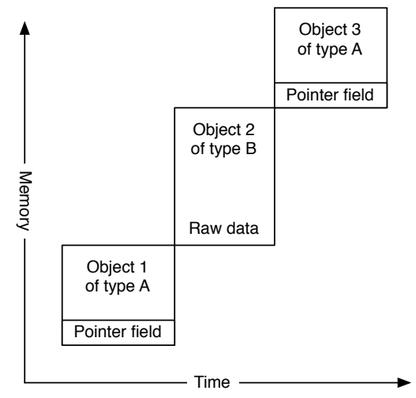



Figure 2: No memory reuse (very safe but expensive).

## 2.2 Naive Defense

A straight forward defense against use-after-free vulnerabilities that takes advantage of the abundant address space of modern 64-bit machines is avoiding any address space reuse. Excessive memory consumption can be avoided by reusing freed memory via the operating system's virtual memory mechanisms (*e.g.* re-

linquishing physical memory using `madvise` with the `MADV_DONTNEED` option on Linux, or other OS specific mechanisms). This simple solution, illustrated in Figure 2, protects against all the attacks discussed in Section 2.1, but has three shortcomings.

First, address space will eventually be exhausted. By then, however, the memory allocator could wrap around and reuse the address space without significant risk.

The second problem is more important. Memory fragmentation limits the amount of physical memory that can be reused through virtual memory mechanisms. Operating systems manage physical memory in units of several Kilobytes in the best case, thus, each small allocation can hold back several Kilobytes of physical memory in adjacent free objects from being reused. In Section 4, we show that the memory overhead of this solution is too high.

Finally, this solution suffers from a high rate of system calls to relinquish physical memory, and attempting to reduce this rate by increasing the block size of memory relinquished with a single system call leads to even higher memory consumption.

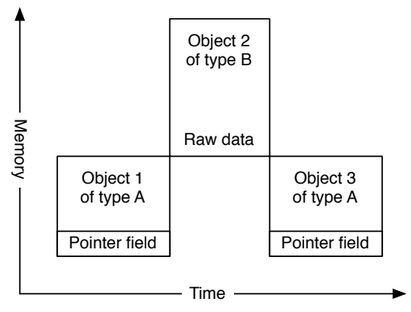



Figure 3: Type-safe memory reuse.

## 2.3 Type-Safe Memory Reuse

Type-safe memory reuse, proposed by Dhurjati *et al.* [9], allows some memory reuse while preserving type safety. It allows dangling pointers, but constrains them to point to objects of the same type and alignment. This way, dereferencing a dangling pointer cannot cause a type violation, rendering use-after-free bugs hard to exploit in practice. As illustrated in Figure 3, with type-safe memory reuse, memory formerly occupied by pointer fields cannot be reused for raw data, preventing attacks as the one in Figure 1.

Moreover, memory formerly occupied by pointer fields can only overlap with the corresponding pointer fields in objects of the same type. This means, for example, that a hijacked function pointer can only be diverted to some other function address used for the same field in a different object, precluding diverting function pointers to attacker injected code, and almost certainly thwarting return-to-libc [20] attacks diverting function pointers to legitimate but suitable executable code in the process. More importantly, objects of the same type share `vtables` and their `vtable` pointers are at the same offsets, thus type-safe memory reuse completely prevents hijacking of `vtable` pointers. This is similar to the attacker constraint discussed in Section 2.1, where the old `vtable` pointer happens to be aligned with another `vtable` pointer, except that attackers are even more constrained now: they cannot exploit differences in inheritance relationships or evade the obstacle by manipulating the heap.

These cases cover generic exploitation techniques and attacks observed in the wild. The remaining attacks are less practical but may be exploitable in some cases, depending on the application and its use of data. Some constraints may still be useful; for example, attacks that hijack data pointers are constrained to only access memory in the corresponding field of another object of the same type. In some cases, this may prevent dangerous corruption or data leakage. However, reusing memory of an object's data fields for another instance of the same type may still enable attacks, including privilege escalation attacks, *e.g.* when data structures holding credentials or access control information for different users are overlapped in time. Another potential exploitation avenue are inconsistencies in the program's data structures that may lead to other memory errors, *e.g.* a buffer may become inconsistent with its size stored in a different object when either is accessed through a dangling pointer. Interestingly, this inconsistency can be detected if spatial protection mechanisms, such as bounds checking, are used in tandem.

# 3 Cling Memory Allocator

The Cling memory allocator is a drop-in replacement for `malloc` designed to satisfy three requirements: *(i)* it does not reuse free memory for its metadata, *(ii)* only allows address space reuse among objects of the same type and alignment, and *(iii)* achieves these without sacrificing performance. Cling combines several solutions from existing memory allocators to achieve its requirements.

## 3.1 Out-of-Band Heap Metadata

The first requirement protects against use-after-free vulnerabilities with dangling pointers to free, not yet reallocated, memory. As we saw in Section 2.1, if the memory allocator uses freed memory for metadata, such as free list pointers, these allocator metadata can be interpreted as object fields, *e.g.* `vtable` pointers, when free memory is referenced through a dangling pointer.

Memory allocator designers have considered using out-of-band metadata before, because attackers targeted in-band heap metadata in several ways: attacker controlled data in freed objects can be interpreted as heap-metadata through double-free vulnerabilities, and heap-based overflows can corrupt allocator metadata adjacent to heap-based buffers. If the allocator uses corrupt heap metadata during its linked list operations, attackers can write an arbitrary value to an arbitrary location.

Although out-of-band heap metadata can solve these problems, some memory allocators mitigate heap metadata corruption without resorting to this solution. For example, attacks corrupting heap metadata can be addressed by detecting the use of corrupted metadata with sanity checks on free list pointers before unlinking a free chunk or using heap canaries [19] to detect corruption due to heap-based buffer overflows. In some cases, corruption can be prevented in the first place, *e.g.* by detecting attempts to free objects already in a free list. These techniques avoid the memory overhead of out-of-band metadata, but are insufficient for preventing use-after-free vulnerabilities, where no corruption of heap metadata takes place.

An approach to address this problem in allocator designs reusing free memory for heap metadata is to ensure that these metadata point to invalid memory if interpreted as pointers by the application. Merely randomizing the metadata by XORing with a secret value may not be sufficient in the face of heap spraying. One option is setting the top bit of every metadata word to ensure it points to protected kernel memory, raising a hardware fault if the program dereferences a dangling pointer to heap metadata, while the allocator would flip the top bit before using the metadata. However, it is still possible that the attacker can tamper with the dangling pointer before dereferencing it. This approach may be preferred when modifying an existing allocator design, but for Cling, we chose to keep metadata out-of-band instead.

An allocator can keep its metadata outside deallocated memory using non-intrusive linked lists (`next` and `prev` pointers stored outside objects) or bitmaps. Non-intrusive linked lists can have significant memory overhead for small allocations, thus Cling uses a two-level allocation scheme where non-intrusive linked lists chain large memory chunks into free lists and small allocations are carved out of buckets holding objects of the same size class using bitmaps. Bitmap allocation schemes have been used successfully in popular memory allocators aiming for performance [10], so they should not pose an inherent performance limitation.

## 3.2 Type-Safe Address Space Reuse

The second requirement protects against use-after-free vulnerabilities where the memory pointed by the dangling pointer has been reused by some other object. As we saw in Section 2.3, constraining dangling pointers to objects within pools of the same type and alignment thwarts a large class of attacks exploiting use-after-free vulnerabilities, including all those used in real attacks. A runtime memory allocator, however, must address two challenges to achieve this. First, it must bridge the semantic gap between type information available to the compiler at compile time and memory allocation requests received at runtime that only specify the number of bytes to allocate. Second, it must address the memory overheads caused by constraining memory reuse within pools. Dhurjati *et al.* [9], who proposed type-safe memory reuse for security, preclude an efficient implementation without using a compile time pointer and region analysis.

To solve the first challenge, we observe that security is maintained even if memory reuse is over-constrained, *i.e.* several allocation pools may exist for the same type, as long as memory reuse across objects of *different* types is prevented. Another key observation is that in C/C++ programs, an allocation site typically allocates objects of a single type or arrays of objects of a single type, which can safely share a pool. Moreover, the allocation site is available to the allocation routines by inspecting their call stack. While different allocation sites may allocate objects of the same type that could also safely share the same pool, Cling's inability to infer this could only affect performance—not security. Section 4 shows that in spite of this pessimization, acceptable performance is achieved.

The immediate caller of a memory allocation routine can be efficiently retrieved from the call stack by inspecting the saved return address. However, multiple tail-call optimizations in a single routine, elaborate control flow, and simple wrappers around allocation routines may obscure the true allocation site. The first two issues are sufficiently rare to not undermine the security of the scheme in general. These problems are elaborated in Section 3.6, and ways to address simple wrappers are described in Section 3.5.

A further complication, illustrated in Figure 4, is caused by array allocations and the lack of knowledge of array element sizes. As discussed, all new objects must be aligned to previously allocated objects, to ensure their fields are aligned one to one. This requirement also applies to array elements. Figure 4, however, illustrates that this constraint can be violated if part of the memory previously used by an array is subsequently reused by an allocation placed at an arbitrary offset relative to

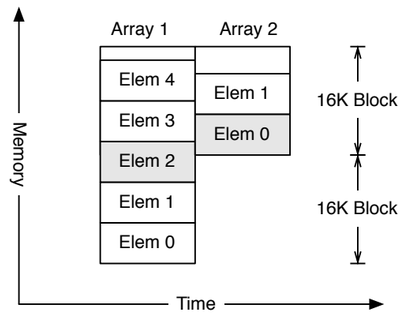



Figure 4: Example of unsafe reuse of array memory, even with allocation pooling, due to not preserving allocation offsets.

the start of the old allocation. Reusing memory from a pool dedicated to objects of the same type is not sufficient for preventing this problem. Memory reuse must also preserve offsets within allocated memory. One solution is to always reuse memory chunks at the same offset within all subsequent allocations. A more constraining but simpler solution, used by Cling, is to allow memory reuse only among allocations of the same size-class, thus ensuring that previously allocated array elements will be properly aligned with array elements subsequently occupying their memory.

This constraint also addresses the variable sized struct idiom, where the final field of a structure, such the following one, is used to access additional, variable size memory allocated at the end of the structure:

```
1 struct {
2     void (*fp)();
3     int len;
4     char buffer[1];
5 };
```

By only reusing memory among instances of such structures that fall into the same size-class, and always aligning such structures at the start of this memory, Cling prevents the structure's fields, *e.g.* the function pointer `fp` in this example, from overlapping after their deallocation with buffer contents of some other object of the same type.

The second challenge is to address the memory overhead incurred by pooling allocations. Dhurjati *et al.* [8] observe that the worst-case memory use increase for a program with $N$ pools would be roughly a factor of $N - 1$: when a program first allocates data of type A, frees all of it, then allocates data of type B, frees all of it, and so on. This situation is even worse for Cling, because it has one pool per size-class per allocation site, instead of just one pool per type.

The key observation to avoid excessive memory overhead is that physical memory, unlike address space, can be safely reused across pools. Cling borrows ideas from previous memory allocators [11] designed to manage physical memory in blocks (via `mmap`) rather than monotonically growing the heap (via `sbrk`). These allocators return individual blocks of memory to the operating system as soon as they are completely free. This technique allows Cling to reuse blocks of memory across different pools.

Cling manages memory in blocks of 16K bytes, satisfying large allocations using contiguous ranges of blocks directly, while carving smaller allocations out of homogeneous blocks called buckets. Cling uses an OS primitive (*e.g.* `madvise`) to inform the OS it can reuse the physical memory of freed blocks.

Deallocated memory accessed through a dangling pointer will either continue to hold the data of the intended object, or will be zero-filled by the OS, triggering a fault if a pointer field stored in it is dereferenced. It is also possible to page protect address ranges after relinquishing their memory (*e.g.* using mechanisms like `mprotect` on top of `madvise`).

Cling does not suffer from fragmentation as the naive scheme described in Section 2.2, because it allows immediate reuse of small allocations' memory within a pool. Address space consumption is also more reasonable: it is proportional to the number of allocation sites in the program, so it does not leak over time as in the naive solution, and is easily manageable in modern 64-bit machines.

## 3.3 Heap Organization

Cling's main heap is divided into blocks of 16K bytes. As illustrated in Figure 5, a smaller address range, dubbed the meta-heap, is reserved for holding block descriptors, one for each 16K address space block. Block descriptors contain fields for maintaining free lists of block ranges, storing the size of the block range, associating the block with a pool, and pointers to metadata for blocks holding small allocations. Metadata for block ranges are only set for the first block in the range—the head block. When address space is exhausted and the heap is grown, the meta-heap is grown correspondingly. The purpose of this meta-heap is to keep heap metadata separate, allowing reuse of the heap's physical memory previously holding allocated data without discarding its metadata stored in the meta-heap.

While memory in the heap area can be relinquished using `madvise`, metadata about address space must be kept in the meta-heap area, thus contributing to the memory overhead of the scheme. This overhead is small. A block descriptor can be under 32 bytes in the current implementation, and with a block size of 16K, this corresponds to memory overhead less than 0.2% of the address space used, which is small enough for the address space usage observed in our evaluation. Moreover, a hashtable could be employed to further reduce this overhead if necessary.

Both blocks and block descriptors are arranged in corresponding linear arrays, as illustrated in Figure 5, so Cling can map between address space blocks and their corresponding block descriptors using operations on their addresses. This allows Cling to efficiently recover the appropriate block descriptor when deallocating memory.

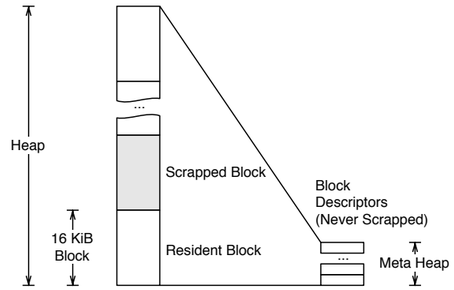



Figure 5: Heap comprised of blocks and meta-heap of block descriptors. The physical memory of deallocated blocks can be scrapped and reused to back blocks in other pools.

Cling pools allocations based on their allocation site. To achieve this, Cling's public memory allocation routines (*e.g.* `malloc` and `new`) retrieve their call site using the return address saved on the stack. Since Cling's routines have complete control over their prologues, the return address can always be retrieved reliably and efficiently (*e.g.* using the `__builtin_return_address` GCC primitive). At first, this return address is used to distinguish between memory allocation sites. Section 3.5 describes how to discover and unwind simple allocation routine wrappers in the program, which is necessary for obtaining a meaningful allocation site in those cases.

Cling uses a hashtable to map allocation sites to pools at runtime. An alternative design to avoid hash table lookups could be to generate a trampoline for each call site and rewrite the call site at hand to use its dedicated trampoline instead of directly calling the memory allocation routine. The trampoline could then call a version of the memory allocation routine accepting an explicit pool parameter. The hash table, however, was preferred because it is less intrusive and handles gracefully corner cases including calling `malloc` through a function pointer. Moreover, since this hash table is accessed frequently but updated infrequently, optimizations such as constructing perfect hashes can be applied in the future, if necessary.

Pools are organized around pool descriptors. The relevant data structures are illustrated in Figure 6. Each pool descriptor contains a table with free lists for block ranges. Each free list links together the head blocks of block ranges belonging to the same size-class (a power of two). These are blocks of memory that have been deallocated and are now reusable only within the pool. Pool descriptors also contain lists of blocks holding small allocations, called buckets. Section 3.4 discusses small object allocation in detail.

Initially, memory is not assigned to any pool. Larger allocations are directly satisfied using a power-of-two range of 16K blocks. A suitable free range is reused from the pool if possible, otherwise, a block range is allocated by incrementing a pointer towards the end of the heap, and it is assigned to the pool. If necessary, the heap is grown using a system call. When these large allocations are deallocated, they are inserted to the appropriate pool descriptor's table of free lists according to their size. The free list pointers are embedded in block descriptors, allowing the underlying physical memory for the block to be relinquished using `madvise`.

## 3.4 Small Allocations

Allocations less than 8K in size (half the block size) are stored in slots inside blocks called buckets. Pool descriptors point to a table with entries to manage buckets for allocations belonging to the same size class. Size classes start from a minimum of 16 bytes, increase by 16 bytes up to 128 bytes, and then increase exponentially up to the maximum of 8K, with 4 additional classes in between each pair of powers-of-two. Each bucket is associated with a free slot bitmap, its element size, and a bump pointer used for fast allocation when the block is first used, as described next.

Using bitmaps for small allocations seems to be a design requirement for keeping memory overhead low without reusing free memory for allocator metadata, so it is critical to ensure that bitmaps are efficient compared to free-list based implementations. Some effort has been put into making sure Cling uses bitmaps efficiently. Cling borrows ideas from reaps [5] to avoid bitmap scanning when many objects are allocated from an allocation site in bursts. This case degenerates to just bumping a pointer to allocate consecutive memory slots. All empty buckets are initially used in bump mode, and

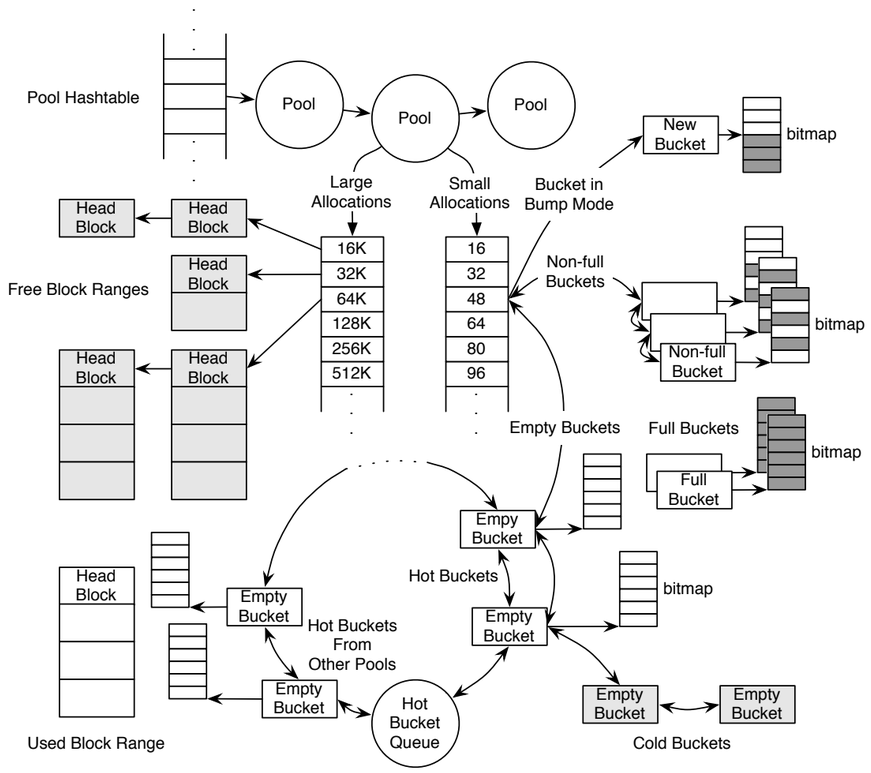



Figure 6: Pool organization illustrating free lists of blocks available for reuse within the pool and the global hot bucket queue that delays reclamation of empty bucket memory. Linked list pointers are not stored inside blocks, as implied by the figure, but rather in their block descriptors stored in the meta-heap. Blocks shaded light gray have had their physical memory reclaimed.

stay in that mode until the bump pointer reaches the end of the bucket. Memory released while in bump mode is marked in the bucket's bitmap but is not used for satisfying allocation requests while the bump pointer can be used.

A pool has at most one bucket in bump mode per size class, pointed by a field of the corresponding table entry, as illustrated in Figure 6. Cling first attempts to satisfy an allocation request using that bucket, if available. Buckets maintain the number of freed elements in a counter. A bucket whose bump pointer reaches the end of the bucket is unlinked from the table entry and, if the counter indicates it has free slots, inserted into a list of non-full buckets. If no bucket in bump mode is available, Cling attempts to use the first bucket from this list, scanning its bitmap to find a free slot. If the counter indicates the bucket is full after an allocation request, the bucket is unlinked from the list of non-full buckets, to avoid obstracting allocations.

Conversely, if the counter of free elements is zero prior to a deallocation, the bucket is re-inserted into the list of non-full buckets. If the counter indicates that the bucket is completely empty after deallocation, it is inserted to a list of empty buckets queuing for memory reuse. This applies even for buckets in bump mode (and was important for keeping memory overhead low). This list of empty buckets is consulted on allocation if there is neither a bucket in bump mode, nor a non-full bucket. If this list is also empty, a new bucket is created using fresh address space, and initialized in bump mode.

Empty buckets are inserted into a global queue of hot buckets, shown at the bottom of Figure 6. This queue has a configurable maximum size (10% of non-empty buckets worked well in our experiments). When the queue size threshold is reached after inserting an empty bucket to the head of the queue, a hot bucket is removed from the tail of the queue, and becomes cold: its bitmap is deallocated, and its associated 16K of memory reused via an `madvise` system call. If a cold bucket is encountered when allocating from the empty bucket list of a pool, a new bitmap is allocated and initialized. The hot bucket queue is important for reducing the number of system calls by trading some memory overhead, controllable through the queue size threshold.

## 3.5 Unwinding Malloc Wrappers

Wrappers around memory allocation routines may conceal real allocation sites. Many programs wrap `malloc` simply to check its return value or collect statistics. Such programs could be ported to Cling by making sure that the few such wrappers call macro versions of Cling's allocation routines that capture the real allocation site, *i.e.* the wrapper's call site. That is not necessary, however, because Cling can detect and handle many such wrappers automatically, and recover the real allocation site by unwinding the stack. This must be implemented carefully because stack unwinding is normally intended for use in slow, error handing code paths.

To detect simple allocation wrappers, Cling initiates a probing mechanism after observing a single allocation site requesting multiple allocation sizes. This probing first uses a costly but reliable unwind of the caller's stack frame (using `libunwind`) to discover the stack location of the suspected wrapper function's return address. Then, after saving the original value, Cling overwrites the wrapper's return address on the stack with the address of a special assembler routine that will be interposed when the suspected wrapper returns. After Cling returns to the caller, and, in turn, the caller returns, the overwritten return address transfers control to the interposed routine. This routine compares the suspected allocation wrapper's return value with the address of the memory allocated by Cling, also saved when the probe was initiated. If the caller appears to return the address just returned by Cling, it is assumed to be a simple wrapper around an allocation function.

To simplify the implementation, probing is aborted if the potential wrapper function issues additional allocation requests before returning. This is not a problem in practice, because simple `malloc` wrappers usually perform a single allocation. Moreover, a more thorough implementation can easily address this.

The probing mechanism is only initiated when multiple allocation sizes are requested from a single allocation site, potentially delaying wrapper identification. It is unlikely, however, that an attacker could exploit this window of opportunity in large programs. Furthermore, this rule helps prevent misidentifying typical functions encapsulating the allocation and initialization of objects of a single type, because these request objects of a single size. Sometimes, such functions allocate arrays of various sizes, and can be misidentified. Nevertheless, these false positives are harmless for security; they only introduce more pools that affect performance by overconstraining allocation, and the performance impact in our benchmarks was small.

Similarly, the current implementation identifies functions such as `strdup` as allocation wrappers. While we could safely pool their allocations (they are of the same type), the performance impact in our benchmarks was again small, so we do not handle them in any special way.

While this probing mechanism handles well the common case of `malloc` wrappers that return the allocated memory through their function return value, it would not detect a wrapper that uses some other mechanism to return the memory, such as modifying a pointer argument passed to the wrapper by reference. Fortunately, such `malloc` wrappers are unusual.

Allocation sites identified as potential wrappers through this probing mechanism are marked as such in the hashtable mapping allocation site addresses to their pools, so Cling can unwind one more stack level to get the real allocation site whenever allocation requests from such an allocation site are encountered, and associate it with a distinct pool.

Stack unwinding is platform specific and, in general, expensive. In 32-bit x86 systems, the frame pointer register `ebp` links stacks frames together, making unwinding reasonably fast, but this register may be re-purposed in optimized builds. Heuristics can still be used with optimized code, *e.g.* looking for a value in the stack that points into the text segment, but they are slower. Data-driven stack unwinding on 64-bit AMD64 systems is more reliable but, again, expensive. Cling uses the `libunwind` library to encapsulate platform specific details of stack unwinding, but caches the stack offset of wrappers' return addresses to allow fast unwinding when possible, as described next, and gives up unwinding if not.

Care must be taken when using a cached stack offset to retrieve the real allocation site, because the cached value may become invalid for functions with a variable frame size, *e.g.* those using `alloca`, resulting in the retrieval of a bogus address. To guard against this, whenever a *new* allocation site is encountered that was retrieved using a cached stack offset, a slow but reliable unwind (using `libunwind`) is performed to confirm the allocation site's validity. If the check fails, the wrapper must have a variable frame size, and Cling falls back to allocating all memory requested through that wrapper from a single

pool. In practice, typical `malloc` wrappers are simple functions with constant frame sizes.

### 3.6 Limitations

Cling prevents `vtable` hijacking, the standard exploitation technique for use-after-free vulnerabilities, and its constraints on function and data pointers are likely to prevent their exploitation, but it may not be able to prevent use-after-free attacks targeting data such as credentials and access control lists stored in objects of a single type. For example, a dangling pointer that used to point to the credentials of one user may end up pointing to the credentials of another user.

Another theoretical attack may involve data structure inconsistencies, when accessed through dangling pointers. For example, if a buffer and a variable holding its length are in separate objects, and one of them is read through a dangling pointer accessing an unrelated object, the length variable may be inconsistent with the actual buffer length, allowing dangerous bound violations. Interestingly, this can be detected if Cling is used in conjunction with a defense offering spatial protection.

Cling relies on mapping allocation sites to object types. A program with contrived flow of control, however, such as in the following example, would obscure the type of allocation requests:

```
1 int size = condition ? sizeof( ↩
    struct A) : sizeof(struct B);
2 void *obj = malloc(size);
```

Fortunately, this situation is less likely when allocating memory using the C++ operator `new` that requires a type argument.

A similar problem occurs when the allocated object is a union: objects allocated at the same program location may still have different types of data at the same offset.

Tail-call optimizations can also obscure allocation sites. Tail-call optimization is applicable when the call to `malloc` is the last instruction before a function returns. The compiler can then replace the call instruction with a simple control-flow transfer to the allocation routine, avoiding pushing a return address to the stack. In this case, Cling would retrieve the return address of the function calling `malloc`. Fortunately, in most cases where this situation might appear, using the available return address still identifies the allocation site uniquely.

Cling cannot prevent unsafe reuse of stack allocated objects, for example when a function erroneously returns a pointer to a local variable. This could be addressed by using Cling as part of a compiler-based solution, by moving dangerous (*e.g.* address taken) stack based variables to the heap at compile time.

Custom memory allocators are a big concern. They allocate memory in huge chunks from the system allocator, and chop them up to satisfy allocation requests for individual objects, concealing the real allocation sites of the program. Fortunately, many custom allocators are used for performance when allocating many objects of a *single* type. Thus, pooling such custom allocator's requests to the system allocator, as done for any other allocation site, is sufficient to maintain type-safe memory reuse. It is also worth pointing that roll-your-own general purpose memory allocators have become a serious security liability due to a number of exploitable memory management bugs beyond use-after-free (invalid frees, double frees, and heap metadata corruption in general). Therefore, using a custom allocator in new projects is not a decision to be taken lightly.

Usability in 32-bit platforms with scarce address space is limited. This is less of a concern for high-end and future machines. If necessary, however, Cling can be combined with a simple conservative collector that scans all words in used physical memory blocks for pointers to used address space blocks. This solution avoids some performance and compatibility problems of conservative garbage collection by relying on information about explicit deallocations. Once address space is exhausted, only memory that is in use needs to be scanned and any 16K block of freed memory that is not pointed by any word in the scanned memory can be reused. The chief compatibility problem of conservative garbage collection, namely hidden pointers (manufactured pointers invisible to the collector), cannot cause premature deallocations, because only explicitly deallocated memory would be garbage collected in this scheme. Nevertheless, relying on the abundant address space of modern machines instead, is more attractive, because garbage collection may introduce unpredictability or expose the program to attacks using hidden dangling pointers.

### 3.7 Implementation

Cling comes as a shared library providing implementations for the `malloc` and the C++ operator `new` allocation interfaces. It can be preloaded with platform specific mechanisms (*e.g.* the LD_PRELOAD environment variable on most Unix-based systems) to override the system's memory allocation routines at program load time.

## 4 Experimental Evaluation

### 4.1 Methodology

We measured Cling's CPU, physical memory, and virtual address space overheads relative to the default GNU libc memory allocator on a 2.66GHz Intel Core 2 Q9400 CPU with 4GB of RAM, running `x86_64` GNU/Linux with a version 2.6 Linux kernel. We also measured two variations of Cling: without wrapper unwinding and using a single pool.

We used benchmarks from the SPEC CPU 2000 and (when not already included in CPU 2000) 2006 benchmark suites [22]. Programs with few allocations and deallocations have practically no overhead with Cling, thus we present results for SPEC benchmarks with at least 100,000 allocation requests. We also used `espresso`, an allocation intensive program that is widely used in memory management studies, and is useful when comparing against related work. Finally, in addition to CPU bound benchmarks, we also evaluated Cling with a current version of the Mozilla Firefox web browser. Web browsers like Firefox are typical attack targets for use-after-free exploits via malicious web sites; moreover, unlike many benchmarks, Firefox is an application of realistic size and running time.

Some programs use custom allocators, defeating Cling's protection and masking its overhead. For these experiments, we disabled a custom allocator implementation in `parser`. The `gcc` benchmark also uses a custom allocation scheme (`obstack`) with different semantics from `malloc` that cannot be readily disabled. We include it to contrast its allocation size distribution with those of other benchmarks. Recent versions of Firefox also use a custom allocator [10] that was disabled by compiling from source with the `--disable-jemalloc` configuration option.

The SPEC programs come with prescribed input data. For `espresso`, we generated a uniformly random input file with 15 inputs and 15 outputs, totalling 32K lines. For Firefox, we used a list of 200 websites retrieved from our browsing history, and replayed it using the `-remote`

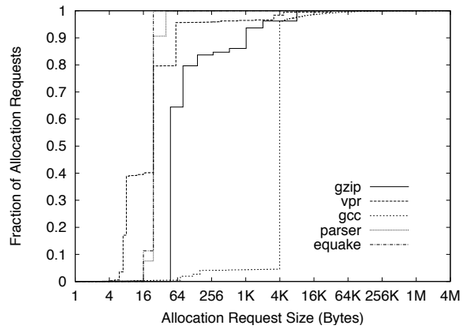



Figure 7: Cumulative distribution function of memory allocation sizes for `gzip`, `vpr`, `gcc`, `parser`, and `equake`.

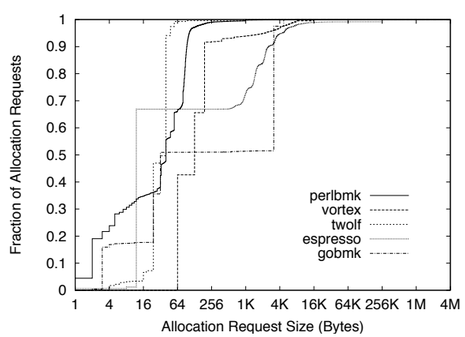



Figure 8: Cumulative distribution function of memory allocation sizes for `perlbmk`, `vortex`, `twolf`, `espresso`, and `gobmk`.

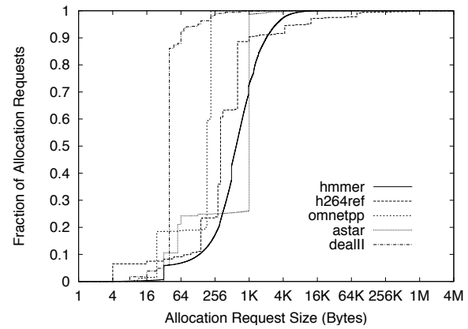



Figure 9: Cumulative distribution function of memory allocation sizes for `hmmer`, `h264ref`, `omnetpp`, `astar`, and `dealII`.

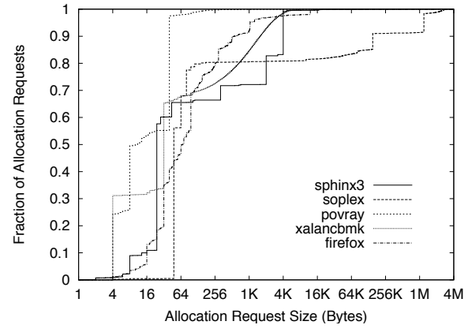



Figure 10: Cumulative distribution function of memory allocation sizes for `sphinx3`, `soplex`, `povray`, `xalancbmk`, and Firefox.

| Benchmark | Allocation Sites | | | Allocation Requests | | Deallocation Requests | |
|---|---|---|---|---|---|---|---|
| | Not Wrappers | Wrappers | Unwound | Small | Large | Small | Large |
| CPU2000 | | | | | | | |
| gzip | 3 | 0 | 0 | 419,724 | 16,483 | 419,724 | 16,463 |
| vpr | 11 | 2 | 59 | 107,184 | 547 | 103,390 | 42 |
| gcc | 5 | 1 | 66 | 194,871 | 4,922 | 166,317 | 4,922 |
| parser | 218 | 3 | 3 | 787,695,542 | 46,532 | 787,523,051 | 46,532 |
| equake | 31 | 0 | 0 | 1,335,048 | 19 | 0 | 0 |
| perlbmk | 10 | 3 | 90 | 31,399,586 | 33,088 | 30,704,104 | 32,732 |
| vortex | 5 | 0 | 0 | 4,594,278 | 28,094 | 4,374,712 | 26,373 |
| twolf | 3 | 1 | 129 | 574,552 | 15 | 492,722 | 5 |
| CPU2006 | | | | | | | |
| gobmk | 50 | 5 | 15 | 621,144 | 20 | 621,109 | 0 |
| hmmer | 8 | 4 | 107 | 2,405,928 | 10,595 | 2,405,928 | 10,595 |
| dealII | 285 | 0 | 0 | 151,324,612 | 7,701 | 151,324,610 | 7,701 |
| sphinx3 | 25 | 2 | 6 | 14,160,472 | 64,086 | 13,959,978 | 63,910 |
| h264ref | 342 | 0 | 0 | 168,634 | 9,145 | 168,631 | 9,142 |
| omnetpp | 158 | 1 | 17 | 267,167,577 | 895 | 267,101,325 | 895 |
| soplex | 285 | 6 | 25 | 190,986 | 44,959 | 190,984 | 44,959 |
| povray | 44 | 0 | 0 | 2,414,082 | 268 | 2,413,942 | 268 |
| astar | 102 | 0 | 0 | 4,797,794 | 2,161 | 4,797,794 | 2,161 |
| xalancbmk | 304 | 1 | 1 | 135,037,352 | 118,205 | 135,037,352 | 118,205 |
| Other | | | | | | | |
| espresso | 49 | 7 | 14 | 3,877,784 | 77,711 | 3,877,783 | 77,711 |
| firefox | 2101 | 51 | 595 | 22,579,058 | 464,565 | 22,255,963 | 464536 |

Table 1: Memory allocation sites and requests in benchmarks and Firefox browser.

option to direct a continuously running Firefox instance under measurement to a new web site every 10 seconds.

We report memory consumption using information obtained through the `/proc/self/status` Linux interface. When reporting physical memory consumption, the sum of the `VmRSS` and `VmPTE` fields is used. The latter measures the size of the page tables used by the process, which increases with Cling due to the larger address space. In most cases, however, it was still very small in absolute value. The `VmSize` field is used to measure address space size. The `VmPeak` and `VmHWM` fields are used to obtain peak values for the `VmSize` and `VmRSS` fields respectively.

The reported CPU times are averages over three runs with small variance. CPU times are not reported for Firefox, because the experiment was IO bound with significant variance.

## 4.2 Benchmark Characterization

Figures 7–10 illustrate the size distribution of allocation requests made by any given benchmark running with their respective input data. We observe that most benchmarks request a wide range of allocation sizes, but the `gcc` benchmark that uses a custom allocator mostly requests memory in chunks of 4K.

Table 1 provides information on the number of static allocation sites in the benchmarks and the absolute number of allocation and deallocation requests at runtime. For allocation sites, the first column is the number of allocation sites that are not wrappers, the second column is the number of allocation sites that are presumed to be in allocation routine wrappers (such as `safe_malloc` in `twolf`, `my_malloc` in `vpr`, and `xmalloc` in `gcc`), and the third column is the number of call sites of these wrappers, that have to be unwound. We observe that Firefox has an order of magnitude more allocation sites than the rest.

The number of allocation and deallocation requests for small (less than 8K) and large allocations are reported separately. The vast majority of allocation requests are for small objects and thus the performance of the bucket allocation scheme is crucial. In fact, no attempt was made to optimize large allocations in this work.

## 4.3 Results

Table 2 tabulates the results of our performance measurements. We observe that the runtime overhead is modest even for programs with a higher rate of allocation and deallocation requests. With the exception of `espresso` (16%), `parser` (12%), and `dealII` (8%), the overhead is less than 2%. Many other benchmarks with few allocation and deallocation requests, not presented here, have even less overhead—an interesting benefit of this approach, which, unlike solutions interposing on memory accesses, does not tax programs not making heavy use of dynamic memory.

In fact, many benchmarks with a significant number of allocations run faster with Cling. For example `xalancbmk`, a notorious allocator abuser, runs 25% faster. In many cases we observed that by tuning allocator parameters such as the block size and the length of the hot bucket queue, we were able to trade memory for speed and vice versa. In particular, with different block sizes, `xalancbmk` would run twice as fast, but with a memory overhead around 40%.



| Benchmark | Execution time | | | | Peak memory usage | | | | Peak VM usage | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Orig. (Sec.) | Cling Ratio | | | Orig. (MiB) | Cling Ratio | | | Orig. (MiB) | Cling Ratio |
| | | Pools | No Unwind | No Pools | | Pools | No Unwind | No Pools | | Pools |
| CPU2000 | | | | | | | | | | |
| gzip | 95.7 | **1.00** | 1.00 | 1.00 | 181.91 | **1.00** | 1.00 | 1.00 | 196.39 | 1.10 |
| vpr | 76.5 | **1.00** | 0.99 | 0.99 | 48.01 | **1.06** | 1.06 | 1.06 | 62.63 | 1.54 |
| gcc | 43.29 | **1.01** | 1.01 | 1.01 | 157.05 | **0.98** | 0.98 | 0.98 | 171.42 | 1.21 |
| parser | 152.6 | **1.12** | 1.08 | 1.05 | 21.43 | **1.14** | 1.13 | 1.05 | 35.99 | 2.26 |
| equake | 47.3 | **0.98** | 1.00 | 0.99 | 49.85 | **0.99** | 0.99 | 0.99 | 64.16 | 1.14 |
| perlbmk | 68.18 | **1.02** | 0.99 | 1.00 | 132.47 | **0.96** | 0.95 | 0.95 | 146.69 | 1.16 |
| vortex | 72.19 | **0.99** | 0.99 | 0.99 | 73.09 | **0.91** | 0.91 | 0.91 | 88.18 | 1.74 |
| twolf | 101.31 | **1.01** | 1.00 | 1.00 | 6.85 | **0.93** | 0.91 | 0.90 | 21.15 | 1.19 |
| CPU2006 | | | | | | | | | | |
| gobmk | 628.6 | **1.00** | 1.0 | 1.00 | 28.96 | **1.01** | 1.00 | 1.00 | 44.69 | 1.64 |
| hmmer | 542.15 | **1.02** | 1.02 | 1.01 | 25.75 | **1.02** | 1.01 | 1.01 | 40.31 | 1.79 |
| dealII | 476.74 | **1.08** | 1.07 | 1.06 | 793.39 | **1.02** | 1.02 | 1.02 | 809.46 | 1.70 |
| sphinx3 | 1143.6 | **1.00** | 1.00 | 0.99 | 43.45 | **1.01** | 1.01 | 1.01 | 59.93 | 1.37 |
| h264ref | 934.71 | **1.00** | 1.01 | 1.01 | 64.54 | **0.97** | 0.97 | 0.96 | 80.18 | 1.52 |
| omnetpp | 573.7 | **0.83** | 0.83 | 0.87 | 169.58 | **0.97** | 0.97 | 0.97 | 183.45 | 1.03 |
| soplex | 524.01 | **1.01** | 1.01 | 1.01 | 421.8 | **1.27** | 1.27 | 1.27 | 639.51 | 2.31 |
| povray | 272.54 | **1.00** | 1.00 | 0.99 | 4.79 | **1.33** | 1.33 | 1.29 | 34.1 | 0.77 |
| astar | 656.09 | **0.93** | 0.93 | 0.92 | 325.77 | **0.94** | 0.94 | 0.94 | 345.51 | 1.56 |
| xalancbmk | 421.03 | **0.75** | 0.75 | 0.77 | 419.93 | **1.03** | 1.03 | 1.14 | 436.54 | 1.45 |
| Other | | | | | | | | | | |
| espresso | 25.21 | **1.16** | 1.07 | 1.10 | 4.63 | **1.13** | 1.06 | 1.02 | 19.36 | 2.08 |

Table 2: Experimental evaluation results for the benchmarks.

In order to factor out the effects of allocator design and tuning as much as possible, Table 2 also includes columns for CPU and memory overhead using Cling with a single pool (which implies no unwinding overhead as well). We observe that in some cases Cling with a single pool is faster and uses less memory than the system allocator, hiding the non-zero overheads of pooling allocations in the full version of Cling. On the other hand, for some benchmarks with higher overhead, such as `dealII` and `parser`, some of the overhead remains even without using pools. For these cases, both slow and fast, it makes sense to compare the overhead against Cling with a single pool. A few programs, however, like `xalancbmk`, use more memory or run slower with a single pool. As mentioned earlier, this benchmark is quite sensitive to allocator tweaks.

Table 2 also includes columns for CPU and memory overhead using Cling with many pools but without unwinding wrappers. We observe that for `espresso` and `parser`, some of the runtime overhead is due to this unwinding.

Peak memory consumption was also low for most benchmarks, except for `parser` (14%), `soplex` (27%), `povray` (33%), and `espresso` (13%). Interestingly, for `soplex` and `povray`, this overhead is not because of allocation pooling: these benchmarks incur similar memory overheads when running with a single pool. In the case of `soplex`, we were able to determine that the overhead is due to a few large `realloc` requests, whose current implementation in Cling is suboptimal. The allocation intensive benchmarks `parser` and `espresso`, on the other hand, do appear to incur memory overhead due to pooling allocations. Disabling unwinding also affects memory use by reducing the number of pools.

The last two columns of Table 2 report virtual address space usage. We observe that Cling's address space usage is well within the capabilities of modern 64-bit machines, with the worst increase less than 150%. Although 64-bit architectures can support much larger address spaces, excessive address space usage would cost in page table memory. Interestingly, in all cases, the address space increase did not prohibit running the programs on 32-bit machines. Admittedly, however, it would be pushing up against the limits.

In the final set of experiments, we ran Cling with Firefox. Since, due to the size of the program, this is the most interesting experiment, we provide a detailed plot of memory usage as a function of time (measured in allocated Megabytes of memory), and we also compare against the naive solution of Section 2.2.

The naive solution was implemented by preventing Cling from reusing memory and changing the memory block size to 4K, which is optimal in terms of memory reuse. (It does increase the system call rate how-

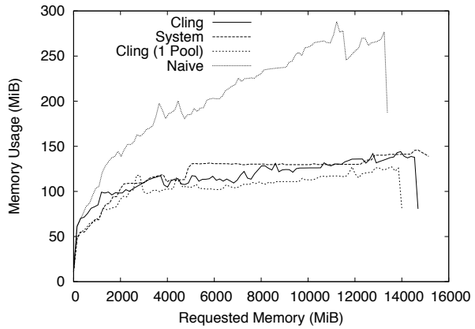



Figure 11: Firefox memory usage over time (measured in requested memory).

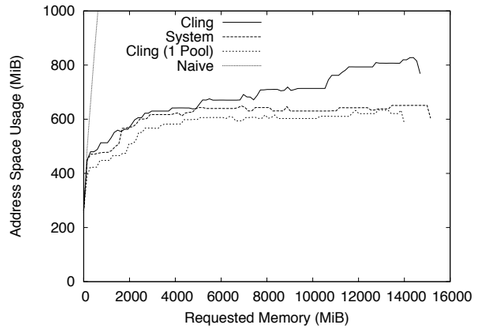



Figure 12: Firefox address space usage over time (measured in requested memory).

ever.) The naive solution could be further optimized by not using segregated storage classes, but this would not affect the memory usage significantly, as the overhead of rounding small allocation requests to size classes in Cling is at most 25%—and much less in practice.

Figure 11 graphs memory use for Firefox. We observe that Cling (with pools) uses similar memory to the system's default allocator. Using pools does incur some overhead, however, as we can see by comparing against Cling using a single pool (which is more memory efficient than the default allocator). Even after considering this, Cling's approach of safe address space reuse appears usable with large, real applications. We observe that Cling's memory usage fluctuates more than the default allocator's because it aggressively returns memory to the operating system. These graphs also show that the naive solution has excessive memory overhead.

Finally, Figure 12 graphs address space usage for Firefox. It illustrates the importance of returning memory to the operating system; without doing so, the scheme's memory overhead would be equal to its address space use. We observe that this implied memory usage with Firefox may not be prohibitively large, but many of the benchmarks evaluated earlier show that there are cases where it can be excessive. As for the address space usage of the naive solution, it quickly goes off the chart because it is linear with requested memory. The naive solution was also the only case where the page table overhead had a significant contribution during our evaluation: in this experiment, the system allocator used 0.99 MiB in page tables, Cling used 1.48 MiB, and the naive solution 19.43 MiB.

## 5 Related Work

Programs written in high-level languages using garbage collection are safe from use-after-free vulnerabilities, because the garbage collector never reuses memory while there is a pointer to it. Garbage collecting unsafe languages like C and C++ is more challenging. Nevertheless, conservative garbage collection [6] is possible, and can address use-after-free vulnerabilities. Conservative garbage collection, however, has unpredictable runtime and memory overheads that may hinder adoption, and is not entirely transparent to the programmer: some porting may be required to eliminate pointers hidden from the garbage collector.

DieHard [4] and Archipelago [16] are memory allocators designed to survive memory errors, including dangling pointer dereferences, with high probability. They can survive dangling pointer errors by preserving the contents of freed objects for a random period of time. Archipelago improves on DieHard by trading address space to decrease physical memory consumption. These solutions are similar to the naive solution of Section 2.2, but address some of its performance problems by eventually reusing memory. Security, however, is compromised: while their probabilistic guarantees are suitable for addressing reliability, they are insufficient against attackers who can adapt their attacks. Moreover, these solutions have considerable runtime overhead for allocation intensive applications. DieHard (without its replication feature) has 12% average overhead but up to 48.8% for `perlbmk` and 109% for `twolf`. Archipelago has 6% runtime overhead across a set of server applications with low allocation rates and few live objects, but the allocation intensive `espresso` benchmark runs 7.32 times slower than using the GNU libc allocator. Cling offers deterministic protection against dangling pointers (but not spatial violations), with significantly lower overhead (*e.g.* 16% runtime overhead for the allocation intensive `espresso` benchmark) thanks to allowing type-safe reuse within pools.

Dangling pointer accesses can be detected using compile-time instrumentation to interpose on every memory access [3, 24]. This approach guarantees complete temporal safety (sharing most of the cost with spatial safety), but has much higher overhead than Cling.

Region-based memory management (*e.g.* [14]) is a language-based solution for safe and efficient memory management. Object allocations are maintained in a lexical stack, and are freed when the enclosing block goes out of scope. To prevent dangling pointers, objects can only refer to other objects in the same region or regions higher up the stack. It may still have to be combined with garbage collection to address long-lived regions. Its performance is better than using garbage collection alone, but it is not transparent to programmers.

A program can be manually modified to use reference-counted smart pointers to prevent reusing memory of objects with remaining references. This, however, requires major changes to application code. HeapSafe [12], on the other hand, is a solution that applies reference counting to legacy code automatically. It has reasonable overhead over a number of CPU bound benchmarks (geometric mean of 11%), but requires recompilation and some source code tweaking.

Debugging tools, such as Electric Fence, use a new virtual page for each allocation of the program and rely on page protection mechanisms to detect dangling pointer accesses. The physical memory overheads due to padding allocations to page boundaries make this approach impractical for production use. Dhurjati *et al.* [8] devised a mechanism to transform memory overhead to address space overhead by wrapping the memory allocator and returning a pointer to a dedicated new virtual page for each allocation but mapping it to the physical page used by the original allocator. The solution's runtime overhead for Unix servers is less than 4%, and for other Unix utilities less than 15%, but incurs up to 11× slowdown for allocation intensive benchmarks.

Interestingly, type-safe memory reuse (dubbed type-stable memory management [13]) was first used to simplify the implementation of non-blocking synchronization algorithms by preventing type errors during speculative execution. In that case, however, it was not applied indiscriminately, and memory could be safely reused after some time bound; thus, performance issues addressed in this work were absent.

Dynamic pool allocation based on allocation site information retrieved by `malloc` through the call stack has been used for dynamic memory optimization [25]. That work aimed to improve performance by laying out objects allocated from the same allocation site consecutively in memory, in combination with data prefetching instructions inserted into binary code.

Dhurjati *et al.* [9] introduced type-homogeneity as a weaker form of temporal memory safety. Their solution uses automatic pool allocation at compile-time to segregate objects into pools of the same type, only reusing memory within pools. Their approach is transparent to the programmer and preserves address space, but relies on imprecise, whole-program analysis.

WIT [2] enforces an approximation of memory safety. It thwarts some dangling pointer attacks by constraining writes and calls through hijacked pointer fields in structures accessed through dangling pointers. It has an average runtime overhead of 10% for SPEC benchmarks, but relies on imprecise, whole-program analysis.

Many previous systems only address the spatial dimension of memory safety (*e.g.* bounds checking systems like [15]). These can be complemented with Cling to address both spatial and temporal memory safety.

Finally, address space layout randomization (ASLR) and data execution prevention (DEP) are widely used mechanisms designed to thwart exploitation of memory errors in general, including use-after-free vulnerabilities. These are practical defenses with low overhead, but they can be evaded. For example, a non-executable heap can be bypassed with, so called, *return-to-libc* attacks [20] diverting control-flow to legitimate executable code in the process image. ASLR can obscure the locations of such code, but relies on secret values, which a lucky or determined attacker might guess. Moreover, buffer over-reads [23] can be exploited to read parts of the memory contents of a process running a vulnerable application, breaking the secrecy assumptions of ASLR.

## 6 Conclusions

Pragmatic defenses against low-level memory corruption attacks have gained considerable acceptance within the software industry. Techniques such as stack canaries, address space layout randomization, and safe exception handling —thanks to their low overhead and transparency for the programmer— have been readily employed by software vendors. In particular, attacks corrupting metadata pointers used by the memory management mechanisms, such as invalid frees, double frees, and heap metadata overwrites, have been addressed with resilient memory allocator designs, benefiting many programs transparently. Similar in spirit, Cling is a pragmatic memory allocator modification for defending against use-after-free vulnerabilities that is readily applicable to real programs and has low overhead.

We found that many of Cling's design requirements could be satisfied by combining mechanisms from suc-

cessful previous allocator designs, and are not inherently detrimental for performance. The overhead of mapping allocation sites to allocation pools was found acceptable in practice, and could be further addressed in future implementations. Finally, closer integration with the language by using compile-time libraries is possible, especially for C++, and can eliminate the semantic gap between the language and the memory allocator by forwarding type information to the allocator, increasing security and flexibility in memory reuse. Nevertheless, the current instantiation has the advantage of being readily applicable to a problem with no practical solutions.

## Acknowledgments

We would like to thank Amitabha Roy for his suggestion of intercepting returning functions to discover potential allocation routine wrappers, Asia Slowinska for fruitful early discussions, and the anonymous reviewers for useful, to-the-point comments.

## References

[1] AFEK, J., AND SHARABANI, A. Dangling pointer: Smashing the pointer for fun and profit. In *Black Hat USA Briefings* (Aug. 2007).

[2] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy* (Los Alamitos, CA, USA, 2008), IEEE Computer Society, pp. 263–277.

[3] AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 1994), ACM, pp. 290–301.

[4] BERGER, E. D., AND ZORN, B. G. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2006), ACM, pp. 158–168.

[5] BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. Reconsidering custom memory allocation. *SIGPLAN Not. 37*, 11 (2002), 1–12.

[6] BOEHM, H.-J., AND WEISER, M. Garbage collection in an uncooperative environment. In *Software Practice & Experience* (New York, NY, USA, 1988), vol. 18, John Wiley & Sons, Inc., pp. 807–820.

[7] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium* (Berkeley, CA, USA, 2005), USENIX Association, pp. 177–192.

[8] DHURJATI, D., AND ADVE, V. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 269–280.

[9] DHURJATI, D., KOWSHIK, S., ADVE, V., AND LATTNER, C. Memory safety without runtime checks or garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES)* (2003), pp. 69–80.

[10] EVANS, J. A scalable concurrent malloc(3) implementation for FreeBSD. BSDCan, Apr. 2006.

[11] FENG, Y., AND BERGER, E. D. A locality-improving dynamic memory allocator. In *Proceedings of the Workshop on Memory System Performance (MSP)* (New York, NY, USA, 2005), ACM, pp. 68–77.

[12] GAY, D., ENNALS, R., AND BREWER, E. Safe manual memory management. In *Proceedings of the 6th International Symposium on Memory Management (ISMM)* (New York, NY, USA, 2007), ACM, pp. 2–14.

[13] GREENWALD, M., AND CHERITON, D. The synergy between non-blocking synchronization and operating system structure. *SIGOPS Oper. Syst. Rev. 30*, SI (1996), 123–136.

[14] GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2002), ACM, pp. 282–293.

[15] JONES, R. W. M., AND KELLY, P. H. J. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging (AADEBUG)* (1997), pp. 13–26.

[16] LVIN, V. B., NOVARK, G., BERGER, E. D., AND ZORN, B. G. Archipelago: trading address space for reliability and security. *SIGOPS Oper. Syst. Rev. 42*, 2 (2008), 115–124.

[17] MITRE CORPORATION. Common vulnerabilities and exposures (CVE). `http://cve.mitre.org`.

[18] MITRE CORPORATION. CWE-416: Use After Free. `http://cwe.mitre.org/data/definitions/416.html`.

[19] ROBERTSON, W., KRUEGEL, C., MUTZ, D., AND VALEUR, F. Run-time detection of heap-based overflows. In *Proceedings of the 17th USENIX Conference on System Administration (LISA)* (Berkeley, CA, USA, 2003), USENIX Association, pp. 51–60.

[20] SOLAR DESIGNER. "return-to-libc" attack. Bugtraq, Aug. 1997.

[21] SOTIROV, A. Heap feng shui in JavaScript. In *Black Hat Europe Briefings* (Feb. 2007).

[22] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC Benchmarks. `http://www.spec.org`.

[23] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., AND WALTER, T. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security (EUROSEC)* (New York, NY, USA, 2009), ACM, pp. 1–8.

[24] XU, W., DUVARNEY, D. C., AND SEKAR, R. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)* (New York, NY, USA, 2004), ACM, pp. 117–126.

[25] ZHAO, Q., RABBAH, R., AND WONG, W.-F. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News 33*, 5 (2005), 27–32.

# ZKPDL: A Language-Based System for Efficient Zero-Knowledge Proofs and Electronic Cash

*Sarah Meiklejohn*
*University of California, San Diego*
smeiklej@cs.ucsd.edu

*C. Chris Erway*
*Brown University*
cce@cs.brown.edu

*Alptekin Küpçü*
*Brown University*
kupcu@cs.brown.edu

*Theodora Hinkle*
*University of Wisconsin, Madison*
thea@cs.wisc.edu

*Anna Lysyanskaya*
*Brown University*
anna@cs.brown.edu

## Abstract

In recent years, many advances have been made in cryptography, as well as in the performance of communication networks and processors. As a result, many advanced cryptographic protocols are now efficient enough to be considered practical, yet research in the area remains largely theoretical and little work has been done to use these protocols in practice, despite a wealth of potential applications.

This paper introduces a simple description language, *ZKPDL*, and an interpreter for this language. ZKPDL implements non-interactive zero-knowledge proofs of knowledge, a primitive which has received much attention in recent years. Using our language, a single program may specify the computation required by both the prover and verifier of a zero-knowledge protocol, while our interpreter performs a number of optimizations to lower both computational and space overhead.

Our motivating application for ZKPDL has been the efficient implementation of electronic cash. As such, we have used our language to develop a cryptographic library, *Cashlib*, that provides an interface for using e-cash and fair exchange protocols without requiring expert knowledge from the programmer.

## 1 Introduction

Modern cryptographic protocols are complicated, computationally intensive, and, given their security requirements, require great care to implement. However, one cannot expect all good cryptographers to be good programmers, or vice versa. As a result, many newly proposed protocols—often described as efficient enough for deployment by their authors—are left unimplemented, despite the potentially useful primitives they offer to system designers. We believe that a lack of high-level software support (such as that provided by OpenSSL, which provides basic encryption and hashing) presents a barrier to the implementation and deployment of advanced cryptographic protocols, and in this work attempt to remove this obstacle.

One particular area of recent cryptographic research which has applications for privacy-preserving systems is *zero-knowledge proofs* [46, 45, 16, 38], which provide a way of proving that a statement is true without revealing anything beyond the validity of the statement. Among the applications of zero-knowledge proofs are electronic voting [48, 55, 37, 50], anonymous authentication [20, 35, 61], anonymous electronic ticketing for public transportation [49], verifiable outsourced computation [8, 42], and essentially any system in which honesty needs to be enforced without sacrificing privacy. Much recent attention has been paid to protocols based on anonymous credentials [29, 34, 23, 25, 10, 7], which allow users to anonymously prove possession of a valid credential (e.g., a driver's license), or prove relationships based on data associated with that credential (e.g., that a user's age lies within a certain range) without revealing their identity or other data. These protocols also prevent the person verifying a credential and the credential's issuer from colluding to link activity to specific users. As corporations and governments move to put an increasing amount of personal information online, the need for efficient privacy-preserving systems has become increasingly important and a major focus of recent research.

Another application of zero-knowledge proofs is electronic cash. The primary aim of our work has been to enable the efficient deployment of secure, anonymous electronic cash (e-cash) in network applications. Like physical coins, e-coins cannot be forged; furthermore, given two e-coins it is impossible to tell who spent them, or even if they came from the same user. For this reason, e-cash holds promise for use in anonymous settings and privacy-preserving applications, where free-riding by users may threaten a system's stability.

Actions in any e-cash system can be characterized as in Figure 1. There are two centralized entities: the bank and the arbiter. The bank keeps track of users' ac-

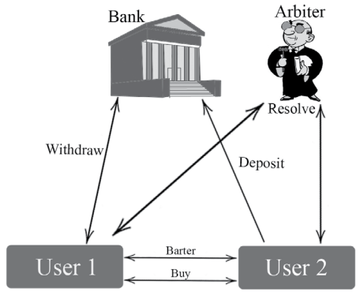



Figure 1: An overview of the entities involved in our e-cash system. Users may engage in buy or barter transactions, withdraw and deposit coins as necessary, and consult the arbiter for resolution only in the case of a dispute.

count balances, lets the users withdraw money, and accepts coin deposits. The arbiter (a trusted third party) resolves any disputes that arise between users in the course of their fair exchanges. Once the users have obtained money from the bank, they are free to exchange coins for items (or just barter for items) and in this way create an economy.

In previous work [9] we describe a privacy-preserving P2P system based on BitTorrent that uses our e-cash and fair exchange protocols to incentivize users to share data. Here, the application of e-cash provides protection against selfish peers, as well as an incentive to upload for peers who have completed their download and thus have no need to continue participating. This system has been realized by our work on the Buy and Barter protocols, described in Section 6.2, which allow a user to fairly exchange e-coins for blocks of data, or barter one block of data for another.

These e-cash protocols can also be used for payments in other systems that face free-riding problems, such as anonymous onion routing [26]. In such a system, routers would be paid for forwarding messages using e-cash, thus providing incentives to route traffic on behalf of others in a manner similar to that proposed by Androulaki et al. [1]. Since P2P systems like these require each user to perform many cryptographic exchanges, the need to provide high performance for repeated executions of these protocols is paramount.

### 1.1 Our contribution

In this paper, we hope to bridge the gap between design and deployment by providing a language, ZKPDL (Zero-Knowledge Proof Description Language), that enables programmers and cryptographers to more easily implement privacy-preserving protocols. We also provide a library, Cashlib, that builds upon our language to provide simple access to cryptographic protocols such as electronic cash, blind signatures, verifiable encryption, and fair exchange.

The design and implementation of our language and library were motivated by collaborations with systems researchers interested in employing e-cash in high-throughput applications, such as the P2P systems described earlier. The resulting performance concerns, and the complexity of the protocols required, motivated our library's focus on performance and ease of use for both the cryptographers designing the protocols and the systems programmers charged with putting them into practice. These twin concerns led to our language-based approach and work on the interpreter.

The high-level nature of our language brings two benefits. First, it frees the programmer from having to worry about the implementation of cryptographic primitives, efficient mathematical operations, generating and processing messages, etc.; instead, ZKPDL allows the specification of a protocol in a manner similar to that of theoretical descriptions. Second, it allows our library to make performance optimizations based on analysis of the protocol description itself.

ZKPDL permits the specification of many widely-used zero-knowledge proofs. We also provide an interpreter that generates and verifies proofs for protocols described by our language. The interpreter performs optimizations such as precomputation of expected exponentiations, translations to prevent redundant proofs, and caching compiled versions of programs to be loaded when they are used again on different inputs. More details on these optimizations are provided in Section 4.2.

Our e-cash library, Cashlib, described in Section 6, sits atop our language to provide simple access to higher-level cryptographic primitives such as e-cash [26], blind signatures [24], verifiable encryption [27], and optimistic fair exchange [9, 51]. Because of the modular nature of our language, we believe that the set of primitives provided by our library can be easily extended to include other zero-knowledge protocols.

Finally, we hope that our efforts will encourage programmers to use (and extend) our library to implement their cryptographic protocols, and that our language will make their job easier; we welcome contribution by our fellow researchers in this effort. Documentation and source code for our library can be found online at `http://github.com/brownie/cashlib`.

## 2 Cryptographic Background

There are two main modern cryptographic primitives used in our framework: *commitment schemes* and *zero-knowledge proofs*. Briefly, a commitment scheme can be thought of as cryptographically analogous to an envelope. When a user Alice wants to commit to a value, she puts the value in the envelope and seals it. Upon receiving a commitment, a second user Bob cannot tell which value is in the envelope; this property is called *hiding* (in this analogy, let's assume Alice is the only one who can open the envelope). Furthermore, because the envelope is sealed, Alice cannot sneak another value into the envelope without Bob knowing: this property is called *binding*. To eventually reveal the value inside the envelope, all Alice has to do is open it (cryptographically, she does this by revealing the private value and any randomness used to form the commitment; this collection of values is aptly referred to as the *opening* of the commitment). We employ both Pedersen commitments [64] and Fujisaki-Okamoto commitments [41, 36], which rely on the security of the Discrete Log assumption and the Strong RSA assumption respectively.

Zero-knowledge proofs [46, 45] provide a way of proving that a statement is true to someone without that person learning anything beyond the validity of the statement. For example, if the statement were "I have access to this sytem" then the verifier would learn only that I really do have access, and not, for example, how I gain access or what my access code is. In our library, we make use of sigma proofs [33], which are three-message proofs that achieve a weaker variant of zero-knowledge known as *honest-verifier zero-knowledge*. We do not implement sigma protocols directly; instead, we use the Fiat-Shamir heuristic [40] that transforms sigma protocols into non-interactive (fully) zero-knowledge proofs, secure in the random oracle model [12].

A primitive similar to zero-knowledge is the idea of a *proof of knowledge* [11], in which the prover not only proves that a statement is true, but also proves that it knows a reason why the statement is true. Extending the above example, this would be equivalent to proving the statement "I have access to the system, and I know a password that makes this true."

In addition to these cryptographic primitives, our library also makes uses of hash functions (both universal one-way hashes [60] and Merkle hashes [59]), digital signatures [47], pseudo-random functions [44], and symmetric encryption [32]. The security of the protocols in our library relies on the security of each of these individual components, as well as the security of any commitment schemes or zero-knowledge proofs used.

## 3 Design

The design of our library and language arose from our initial goal of providing a high-performance implementation of protocols for e-cash and fair exchange for use in applications such as those described in the introduction. For these applications, the need to support many repeated interactions of the same protocol efficiently is a paramount concern for both the bank and the users. In the bank's case, it must conduct withdraw and deposit protocols with every user in the system, while in the user's case it is possible that a user would want to conduct many transactions using the same system parameters.

Motivated by these performance requirements, we initially developed a more straightforward implementation of our protocols using C++ and GMP [43], but found that our ability to modify and optimize our implementation was hampered by the complexity of our protocols. High-level changes to protocols required significant effort to re-implement; meanwhile, potentially useful performance optimizations became difficult to implement, and there was no way to easily extend the functionality of the library.

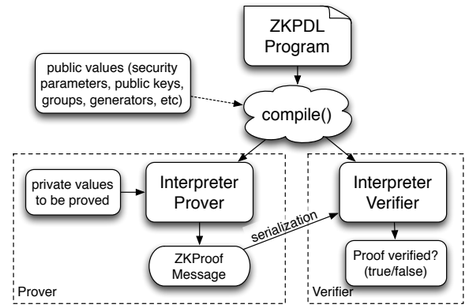



Figure 2: Usage of a ZKPDL program: the same program is compiled separately by the prover and verifier, who may also be provided with a set of fixed public parameters. This produces an Interpreter object, which can be used by the prover to prove to a verifier that his private values satisfy a certain set of relationships. Serialization and processing of proof messages are provided by the library. Once compiled, an interpreter can be re-used on different private inputs, using the same public parameters that were originally provided.

These difficulties led to our current design, illustrated in Figure 2. Our system allows a pseudocode-like description of a protocol to be developed using our description language, ZKPDL. This program is compiled by our interpreter, and optionally provided a list of public parameters, which are "compiled in" to the program. At compile time, a number of transformations and optimizations are performed on the abstract syntax tree produced by our parser, which we developed using the ANTLR parser generator [63]. Once compiled, these interpreter objects can be used repeatedly by the prover to generate zero-knowledge proofs about private values, or by the

verifier to verify these proofs.

Key to our approach is the simplicity of our language. It is not Turing-complete and does not allow for branching or conditionals; it simply describes the variables, equations, and relationships required by a protocol, leaving the implementation details up to the interpreter and language framework. This framework, described in the following section, provides C++ classes that parse, analyze, optimize, and interpret ZKPDL programs, employing many common compiler techniques (e.g., constant substitution and propagation, type-checking, providing error messages when undefined variables are used, etc.) in the process. We are able to understand and transform mathematical expressions into forms that provide better performance (e.g., through techniques for fixed-base exponentiation), and recognize relationships between values to be proved in zero-knowledge. All of these low-level optimizations, as well as our high-level primitives, should enable a programmer to quickly implement and evaluate the efficiency of a protocol.

We also provide a number of C++ classes that wrap ZKPDL programs into interfaces for generating and verifying proofs, as well as marshaling them between computers. We build upon these wrappers to additionally provide Cashlib, a collection of interfaces that allows a programmer to assume the role of buyer, seller, bank, or arbiter in a fair exchange system based on endorsed e-cash [26], as seen in Figure 1 and described in Section 5.3.

# 4 Implementation of ZKPDL

To enable implementation of the cryptographic primitives discussed in Section 2, we have designed a programming language for specifying zero-knowledge protocols, as well as an interpreter for this language. The interpreter is implemented in C++ and consists of approximately 6000 lines of code. On the prover side, the interpreter will output a zero-knowledge proof for the relations specified in the program; on the verifier side, the interpreter will be given a proof and verify whether or not it is correct. Therefore, the output of the interpreter depends on the role of the user, although the program provided to the interpreter is the same for both.

## 4.1 Overview

Here we provide a brief overview of some fundamental language features to give an idea of how programs are written; a full grammar for our language, containing all of its features, can be found in our documentation available online, and further sample programs can be found in Section 5. A program can be broken down into two blocks: a computation block and a proof block. Each of these blocks is optional: if a user just wants a calculator for modular (or even just integer) arithmetic then he will specify just the computation block; if, on the other hand, he has all the input values pre-computed and justs wants a zero-knowledge proof of relations between these values, he will specify just the proof block. Here is a sample program written in our language (indentations are included for readability, and are not required syntax).

sample.zkp

```
1  computation: // compute values required for proof
2    given: // declarations
3      group: G = <g,h>
4      exponents in G: x[2:3]
5    compute: // declarations and assignments
6      random exponents in G: r[1:3]
7      x_1 := x_2 * x_3
8      for(i, 1:3, c_i := g^x_i * h^r_i)
9
10 proof:
11   given: // declarations of public values
12     group: G = <g,h>
13     elements in G: c[1:3]
14     for(i, 1:3, commitment to x_i: c_i = g^x_i * h^r_i)
15   prove knowledge of: // declarations of private values
16     exponents in G: x[1:3], r[1:3]
17   such that: // protocol specification; i.e. relations
18     x_1 = x_2 * x_3
```

In this example, we are proving that the value $x_1$ contained within the commitment $c_1$ is the product of the two values $x_2$ and $x_3$ contained in the commitments $c_2$ and $c_3$. The program can be broken down in terms of how variables are declared and used, and the computation and proof specifications. Note that some lines are repeated across the computation and proof blocks, as both are optional and hence considered independently.

### 4.1.1 Variable declaration

Two types of variables can be declared: group objects and numerical objects. Names of groups must start with a letter and cannot have any subscripts; sample group declarations can be seen in lines 3 and 12 of the above program. In these lines, we also declare the group generators, although this declaration is optional (as we will see later on in Section 5, it is also optional to name the group modulus).

Numerical objects can be declared in two ways. The first is in a list of variables, where their type is specified by the user. Valid types are `element`, `exponent` (which refer respectively to elements within a finite-order group and the corresponding exponents for that group), and `integer`; it should be noted that for the first two of these types a corresponding group must also be specified in the type information (see lines 4 and 13 for an example). The other way in which variables can be declared is in the compute block, where they are declared as they are being assigned (meaning they appear on the left-hand side of an equation), which we can see in lines 7 and 8. In this case, the type is inferred by the values on the right-hand side of the equation; a compile-time exception will be thrown if the types do not match up (for example, if elements from two different groups are being multiplied).



Numerical variables must start with a letter and are allowed to have subscripts.

### 4.1.2 Computation

The computation block breaks down into two blocks of its own: the `given` block and the `compute` block. The `given` block specifies the parameters, as well as any values that have already been computed by the user and are necessary for the computation (in the example, the group `G` can be considered a system parameter and the values `x_2` and `x_3` are just needed for the computation).

The `compute` block carries out the specified computations. There are two types of computations: picking a random value, and defining a value by setting it equal to the right-hand side of an equation. We can see an example of the former in line 6 of our sample program; in this case, we are picking three random exponents in a group (note `r[1:3]` is just syntactic sugar for writing `r_1, r_2, r_3`). We also support picking a random integer from a specified range, and picking a random prime of a specified length (examples of these can be found in Section 5). As already noted, lines 7 and 8 provide examples of lines for computing equations. In line 8, the `for` syntax is again just syntactic sugar; this time to succintly specify the relations `c_1 = g^x_1*h^r_1`, `c_2 = g^x_2*h^r_2`, and `c_3 = g^x_3*h^r_3`. We have a similar `for` syntax for specifying products or sums (much like $\prod$ or $\sum$ in conventional mathematical notation), but neither of these `for` macros should be confused with a for loop in a conventional programming language.

### 4.1.3 Proof specification

The proof block is comprised of three blocks: the `given` block, the `prove knowledge of` block, and the `such that` block. In the `given` block, the parameters for the proof are specified, as well as the public inputs known to both the prover and verifier for the zero-knowledge protocol. In the `prove knowledge of` block, the prover's private inputs are specified. Finally, the `such that` block specifies the desired relations between all the values; the zero-knowledge proof will be a proof that these relations are satisfied. We currently support four main types of relations:

- Proving knowledge of the opening of a commitment [66]. We can prove openings of Pedersen [64] or Fujisaki-Okamoto commitments [41, 36]. In both cases we allow for commitments to multiple values.
- Proving equality of the openings of different commitments. Given any number of commitments, we can prove the equality of any subset of the values contained within the commitments.
- Proving that a committed value is the product of two other committed values [36, 17]. As seen in our sample program, we can prove that a value $x$ contained within a commitment is the product of two other values $y, z$ contained within two other commitments; i.e., $x = y \cdot z$. As a special case, we can also prove that $x = y^2$.
- Proving that a committed value is contained within a public range [17, 54]. We can prove that the value $x$ contained within a given commitment satisfies $lo \leq x < hi$, where $lo$ and $hi$ are both public values.

There are a number of other zero-knowledge proof types (e.g., proving a value is a Blum integer, proving that committed values satisfy some polynomial relationship, etc.), but we chose these four based on their wide usage in applications, in particular in e-cash and anonymous credentials. We note, however, that adding other proof types to the language should require little work (as mentioned in Section 4.2), as we specifically designed the language and interpreter with modularity in mind.

### 4.1.4 Sample usage

In addition to showing a sample program, we would also like to demonstrate a sample usage of our interpreter API. In order to use the sample ZKPDL program from Section 4.1, one could use the following C++ code (assuming there are already numerical variables named `x2` and `x3`, and a group named `G`):

```
group_map g;
variable_map v;
g["G"] = G;
v["x_2"] = x2;
v["x_3"] = x3;
InterpreterProver prover;
// compiles program with groups
prover.check("sample.zkp", g);
// computes intermediate values needed for proof
prover.compute(v);
// computes and outputs proof
ProofMessage proofMsg = (prover.getPublicVariables(),
                         prover.computeProof());
```

The method is the same for all programs: any necessary groups and/or variables are inserted into the appropriate maps, which are then passed to the interpreter. Note that the group map in this case is passed to the interpreter at "compile time" so that it may pre-compute powers of group generators to be used for exponentiation optimizations (described in the next section); however, both the group and variable maps may be provided at "compute time." Any syntactic errors will be caught at compile time, but if the inputs provided at compute time are not valid for the relations being proved, the proof will be computed anyway and the error will be caught by the verifier. The `ProofMessage` is a serializable container

for the zero-knowledge proof and any intermediate values (e.g., commitments and group bases) that the verifier might need to verify the proof.

The method is almost identical for the verifier:

```
group_map g;
variable_map v;
g["G"] = G;
InterpreterVerifier verifier;
verifier.check("sample.zkp", g);
verifier.compute(v, proofMsg.publics);
bool verified = verifier.verify(proofMsg.proof);
```

As we can see, the main difference is that the verifier uses both its own public inputs and the prover's public values at compute time (with its own inputs always taking precedence over the `ProofMessage` inputs), but still takes in the proof to be checked afterwards so that the actions of the prover and verifier remain symmetric.

## 4.2 Optimizations

In our interpreter, we have incorporated a number of optimizations that make using our language not only more convenient but also more efficient. Here we describe the most significant optimizations, which include removing any redundancy when multiple proofs are combined and performing multi-exponentiations on cached bases when the same bases are used frequently. Other improvements specific to existing protocols can be found in Section 5.

### 4.2.1 Translation

To eliminate redundancy between different proofs, we first translate each proof described in Section 4.1.3 into a "fundamental discrete logarithm form." In this form, each proof can be represented by a collection of equations of the form $A = B^x \cdot C^y$. For example, if the prover would like to prove that the value $x$ contained within $C_x = g^x h^{r_x}$ is equal to the product of the values $y$ and $z$ contained within $C_y = g^y h^{r_y}$ and $C_z = g^z h^{r_z}$ respectively, this is equivalent to a proof of knowledge of the discrete logarithm equalities $C_y = g^y h^{r_y}$ and $C_x = C_y^z h^{r_x - z r_y}$.

Our sample program in the previous section is first translated into this discrete logarithm form. During runtime, the values provided to the prover are then used to generate the zero-knowledge proof. In addition to eliminating redundancy between proofs of different relations in the program, this technique also allows our language to easily add new types of proofs as they become available. To add any proof that can be broken down into this discrete logarithm form, we need to add only a translation function and a rule in the grammar for how we would like to specify this proof in a program, and the rest of the work will be handled by our existing framework.

### 4.2.2 Multi-exponentiation

The computational performance of many cryptographic protocols, especially those used by our library, is often dominated by the need to perform many modular exponentiation operations on large integers. These operations typically involve the use of systems parameters as bases, with exponents chosen at random or provided as private inputs (e.g., Pedersen commitments, which require computation of $g^x \cdot h^r$, where $g$ and $h$ are publicly known). Algorithms for simultaneous multiple exponentiation allow the result of multi-base exponentiations such as these to be computed without performing each intermediate exponentiation individually; an overview of these protocols can be found in Section 14.6 of Menezes et al. [58].

Our interpreter leverages the descriptions of mathematical expressions in ZKPDL programs to recognize when fixed-base exponentiation operations occur, allowing it to precompute lookup tables at compile time that can speed up these computations dramatically. In addition to single-table multi-exponentiation techniques (i.e., the $2^w$-ary method [58]), we offer programmers who expect to run the same protocol many times the ability to take advantage of time/space tradeoffs by generating large lookup tables of precomputed powers. This allows a programmer to choose parameters that balance the memory requirements of the interpreter against the need for fast exponentiation.

For single-base exponentiation, we employ window-based precomputation techniques similar to those used by PBC [56] to cache powers of fixed bases. For multi-base exponentiation of $k$ exponents, we currently extend the $2^w$-ary method to store $2^{kw}$-sized lookup tables for each $w$-bit window of the expected exponent length, so that multi-exponentiations on exponents of length $n$ require only $n/w$ multiplications of stored values. While we are also evaluating other algorithms offering similar time-space tradeoffs, we demonstrate the performance gains afforded by these techniques later in Table 1.

### 4.2.3 Interpreter caching

We also cache the parsed, compiled environments of ZKPDL programs when they are first run. Because we accept system parameters at compile time, we are able to evaluate and propagate any subexpressions made up of fixed constants and perform exponentiation precomputations before these expressions are fully evaluated at runtime. Even without the use of large tables for fixed-based exponentiation, this optimization proves useful when repeated executions of the same program must be performed; e.g., for a bank dealing with e-coin deposits. In this case, a bank must invoke the interpreter for each coin deposited; looking ahead to Table 1 we see that, on average, this operation takes the bank 83ms. If our program were re-parsed each time, it would take an extra 10ms, as opposed to the fraction of a millisecond required to load a cached interpreter environment, saving the bank approximately 10% of computation time per transaction by avoiding parsing overhead.



# 5 Sample Programs and Performance

Using our language, we have written programs for a wide variety of cryptographic primitives, including blind signatures [24], verifiable encryption [27], and endorsed e-cash [26]. In the following sections, we provide our programs for these three primitives; in addition, performance benchmarks for all of them can be found at the end of the section.

## 5.1 CL signatures

Using our language, we have implemented the blind signature scheme due to Camenisch and Lysyanskaya [24]; as we will see in Section 5.3, CL signatures are integral to endorsed e-cash. Briefly, a *blind signature*, as introduced by Chaum [28], enables a signature issuer to sign a message without learning the contents of the message. A CL signature works in two main phases: an issuing phase, in which a user actually obtains the signature, and a proving phase, in which the user is able to prove (in zero-knowledge) to other users that he does in fact possess a valid CL signature.

The issuing phase is a one-round interaction between the recipient and the issuer, at the end of which the recipient obtains the blind signature on her message(s). Because the protocol is interactive, we present one program for each stage of the protocol. At the end of this first stage, the signature issuer will return a *partial signature* to the recipient, who will then use this signature to compute the full signature on the hidden message.

cl-recipient-proof.zkp

```
computation:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    exponents in pkGroup: x[1:L]
    integers: stat, modSize
  compute:
    random integer in [0,2^(modSize+stat)): vprime
    C := hprime^vprime * for(i, 1:L, *, gprime_i^x_i)

proof:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    group: comGroup = <f, g, h, h1, h2>
    element in pkGroup: C
    elements in comGroup: c[1:L]
       for(i, 1:L, commitment to x_i: c_i=g^x_i*h^r_i)
    integer: l_x
  prove knowledge of:
    integers: x[1:L]
    exponents in comGroup: r[1:L]
    exponent in pkGroup: vprime
  such that:
    for(i, 1:l, range: (-(2^l_x-1)) <= x_i < 2^l_x)
    C = hprime^vprime * for(i, 1:L, gprime_i^x_i)
    for(i, 1:L, c_i = g^x_i * h^r_i)
```

Next, the issuer must prove the partial signature is computed correctly, as in the following program.

cl-issuer-proof.zkp

```
computation:
  given:
    group: pkGroup = <f, g[1:L+k], h>
    element in pkGroup: C
    exponents in pkGroup: x[1:k+L]
    integers: stat, modSize, lx
  compute:
    random integer in [0,2^(modSize+lx+stat)): vpp
    random prime of length lx+2: e
    einverse := 1/e
    A := (f*C*h^vpp * for(i,L+1:k+L,*,g_i^x_i))^einverse

proof:
  given:
    group: pkGroup = <f, g[1:L+k], h>
    elements in pkGroup: A, C
    exponents in pkGroup: e, vpp, x[L+1:k]
  prove knowledge of:
    exponents in pkGroup: einverse
  such that:
    A = (f*C*h^vpp * for(i,L+1:k+L,*,g_i^x_i))^einverse
```

Once the recipient obtains the partial signature, she can unblind it to obtain a full signature; this step completes the issuing phase.

Now, the owner of a CL signature needs a way to prove that she has a signature, without revealing either the signature or the values. To accomplish this, the prover first randomizes the CL signature and then attaches a zero-knowledge proof of knowledge that the randomized signature corresponds to the original signature on the committed message.

cl-possession-proof.zkp

```
computation:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    element in pkGroup: A
    exponents in pkGroup: e, v, x[1:L]
    integers: modSize, stat
  compute:
    random integers in [0,2^(modSize+stat)): r, r_C
    vprime := v + r*e
    Aprime := A * hprime^r
    C := h^r_C * for(i, 1:L, *, gprime_i^x_i)
    D := for(i, L+1:L+k, *, gprime_i^x_i)
    fCD := f * C * D

proof:
  given:
    group: pkGroup = <fprime, gprime[1:L+k], hprime>
    group: comGroup = <f, g, h, h1, h2>
    elements in pkGroup: C, D, Aprime, fCD
    elements in comGroup: c[1:L]
       for(i, 1:L, commitment to x_i: c_i=g^x_i*h^r_i)
    exponents in pkGroup: x[L+1:L+k]
    integer: l_x
  prove knowledge of:
    integers: x[1:L]
    exponents in comGroup: r[1:L]
    exponents in pkGroup: e, vprime, r_C
  such that:
    for(i, 1:L, range: (-(2^l_x - 1)) <= x_i < 2^l_x)
    C = hprime^r_C * for(i, 1:l, *, gprime_i^x_i)
    for(i, 1:L, c_i = g^x_i * h^r_i)
    fCD = (Aprime^e) * hprime^(r_C - vprime)
```

## 5.2 Verifiable encryption

Briefly, verifiable encryption consists of a ciphertext under the public key of some trusted third party (in our case, the arbiter) and a zero-knowledge proof that the values inside the ciphertext satisfy some relation; this pair is often referred to as a *verifiable escrow*. Our implementation of verifiable encryption is based on the construction of Camenisch and Shoup [27]. The main use of verifiable encryption in e-cash is to allow a user to verifiably encrypt the opening of a commitment under the public key of the arbiter. A recipient of such a verifiable escrow can then verify that the encrypted values correspond to the opening of the commitment.

verifiable-encryption.zkp

```
computation:
  given:
    group: secondGroup = <g[1:m], h>
    group: RSAGroup
    modulus: N
    group: G
    group: cashGroup = <f_3, gprime, hprime, f_1, f_2>
    exponents in G: x[1:m]
    elements in G: u[1:m], v, w
  compute:
    random integer in [0,N/4): s
    random exponents in secondGroup: r[1:m]
    for(i, 1:m, c_i := g_1^x_i * g_2^r_i)
    Xprime := for(i, 1:m, *, g_i^x_i) * h^s
    vsquared := v^2
    wsquared := w^2
    for(i, 1:m, usquared_i := u_i^2)

proof:
  given:
    group: secondGroup = <g[1:m], h>
    group: G
    group: RSAGroup
    modulus: N
    group: cashGroup = <f_3, gprime, hprime, f_1, f_2>
    element in cashGroup: X
    elements in secondGroup: Xprime, c[1:m]
    for(i,1:m,commitment to x_i: c_i=g_1^x_i*g_2^r_i)
    elements in G: a[1:m], b, d, e, f, usquared[1:m],
                   vsquared, wsquared
  prove knowledge of:
    integers: x[1;M], r
    exponent in G: hash
    exponents in secondGroup: r[1:m], s
  such that:
    for(i, 1:m, range: -N/2 + 1 <= x_i < N/2)
    vsquared = f^(2*r)
    wsquared = (d * e^hash)^(2*r)
    for(i, 1:m, usquared_i = b^(2*x_i) * a_i^(2*r))
    X = for(i, 1:m, *, f_i^x_i)
    Xprime = for(i, 1:m, *, g_i^x_i) * h^s
```

## 5.3 E-cash

Electronic cash, or e-cash for short, was first introduced by Chaum [28] and can be thought of as the electronic equivalent of cash; i.e., an electronic currency that preserves users' anonymity, as opposed to electronic checks [30] or credit cards. We implement endorsed e-cash, due to Camenisch, Lysyanskaya, and Meyerovich [26] (which is an extension of compact e-cash [21]), for two main reasons. Our first reason is that an endorsed e-coin can be split up into two parts, its endorsement and an unendorsed component; only with both of these parts can the coin be considered complete. As we will see in Section 6.2.1, this property enables efficient fair exchange. The second reason for choosing endorsed e-cash is that it is *offline*, which means the bank does not need to be active in every transaction; this significantly reduces the burden placed on the bank. Although the bank does not check the coin in every interaction, endorsed e-cash has the property that double-spenders (i.e., users who try to spend the same coin twice) can be caught by the bank at the time of deposit and punished accordingly. Because e-cash is meant to preserve privacy, however, a user is also guaranteed that unless she double spends a coin, her identity will be kept secret.

During the withdrawal phase of endorsed e-cash, a user contacts the bank. Before withdrawing, the user will have registered with the bank by storing a commitment. In order to prove her identity, then, the user will provide a proof that she knows the opening of the registered commitment. This can be accomplished using the following simple program:

user-id-proof.zkp

```
proof:
  given:
    group: cashGroup = <f,g,h,h1,h2>
    elements in cashGroup: A, pk_u
      commitment to sk_u: A = g^sk_u * h^r_u
  prove knowledge of:
    exponents in cashGroup: sk_u, r_u
  such that:
    pk_u = g^sk_u
    A = g^sk_u * h^r_u
```

Once the bank has verified this proof, the user and the bank will run a protocol to obtain a CL signature (using the programs we saw in Section 5.1) on the user's identity and two pseudo-random function seeds. These private values and the signature on them define a wallet that contains $W$ coins (where $W$ is a system-wide public parameter).

When a user wishes to spend one of her coins, she splits it up into its unendorsed part and the endorsement. She then sends the unendorsed component to a merchant and proves it is valid. If the merchant then sends her what she wanted to buy, she will follow up with the endorsement to complete the coin and the transaction is complete. The following program is used for proving the validity of a coin.

coin-proof.zkp

```
computation:
  given:
    group: cashGroup = <f, g, h, h1, h2>
    exponents in cashGroup: s, t, sk_u
    integer: J
  compute:
    random exponents in cashGroup: r_B, r_C, r_D, x1,
                                   x2, r_y, R
    alpha := 1 / (s + J)
    beta := 1 / (t + J)
    C := g^s * h^r_C
    D := g^t * h^r_D
    y := h1^x1 * h2^x2 * f^r_y
    B := g^sk_u * h^r_B
    S := g^alpha * g^x1
    T := g^sk_u * (g^R)^beta * g^x2

proof:
  given:
    group: cashGroup = <f, g, h, h1, h2>
    elements in cashGroup: y, S, T, B, C, D
       commitment to sk_u: B = g^sk_u * h^r_B
       commitment to s: C = g^s * h^r_C
       commitment to t: D = g^t * h^r_D
    integer: J
  prove knowledge of:
    exponents in cashGroup: x1, x2, r_y, sk_u, alpha,
          beta, s, t, r_B, r_C, r_D, R
  such that:
    y = h1^x1 * h2^x2 * f^r_y
    S = g^alpha * g^x1
    T = g^sk_u * (g^R)^beta * g^x2
    g = (g^J * C)^alpha * h^(-r_C / (s+J))
    g = (g^J * D)^beta * h^(-r_D / (t+J))
```

## 5.4 Performance

Here we measure the communication and computational resources used by our system when running each of the programs above. The benchmarks presented in Table 1 were collected on a MacBook Pro with a 2.53GHz Intel Core 2 Duo processor and 4GB of RAM running OS X 10.6; we therefore expect that these results will reflect those of a typical home user with no special cryptographic hardware support.

As for speed, caching exponents of fixed bases results in a significant performance increase, making it an important optimization for applications that require repeated protocol executions. The only caveat is that the exponent cache required for complex protocols can grow to hundreds of megabytes (using faster-performing parameters), and so our library allows users to choose whether to use caching, and if so how much of the cache should be used by this optimization.

The time taken for the higher-level protocols provides a clear view of the complexity of each protocol. For example, the marked difference between the time required to generate a CL issuer proof and a CL possession proof can be attributed to the fact that a CL issuer proof requires proving only one discrete log relation, while a CL possession proof on three private values requires three range proofs and five more discrete log relations.

Table 1 also shows that verifiable encryption is by far the biggest bottleneck, requiring almost three times as much time to compute as any other step. As seen in the program in Section 5.2, there is one range proof performed for each value contained in the verifiable escrow. In order to perform a range proof, the value contained in the range must be decomposed as a sum of four squares [65]. Because the values used in our verifiable encryption program are much larger than the ones used in CL signatures (about 1024 vs. 160 bits, to get 80-bit security for both), this decomposition often takes considerably more time for verifiable encryption than it does for CL signatures. Furthermore, since the values being verifiably encrypted are different each time, caching the decomposition of the values wouldn't be of any use.

A final observation on computational performance is that proving possession of a CL signature completely dominates the time required to prove the validity of a coin, since the timings for the two proofs are within milliseconds. This suggests that the only way to significantly improve the performance of e-coins and verifiable encryption would be to develop more efficient techniques for range proofs (which has in fact been the subject of some recent cryptographic research [48, 18, 67]).

In terms of proof size, range proofs are much larger than proofs for discrete logarithms or multiplication. This is to be expected, as translating a range proof into discrete logarithm form (as described in Section 4.2) requires eleven equations, whereas a single DLR proof requires only one, and a multiplication proof requires two.

# 6 Implementation of Cashlib

Using the primitives described in the previous section, we wrote a cryptographic library designed for optimistic fair exchange protocols. Fair exchange [31] involves a situation in which a buyer wants to make sure that she doesn't pay a merchant unless she gets what she is buying, while the merchant doesn't want to give away his goods unless he is guaranteed to be paid. It is known that fair exchange cannot be done without a trusted third party [62], but *optimistic* fair exchange [2, 3] describes the cases in which the trusted third party has to get involved only in the case of a dispute.

The library was written in C++ and consists of approximately 11000 lines of code in addition to the interpreter. A previous version of the library in which all the protocols and proofs were hand-coded (i.e., the interpreter was not used) consisted of approximately 20000 lines of code, meaning that the use of roughly 400 lines of ZKPDL was able to replace 9000 lines of our original C++ code (and, as we will see, make our operations more efficient as well).

## 6.1 Endorsed e-cash

A description of endorsed e-cash can be found in Section 5.3; the version used in our library, however, contains a number of optimizations. Just as with real cash, we now allow for different coin denominations. Each coin denomination corresponds to a different bank public key, so once the user requests a certain denomination, the wallet is then signed using the corresponding public key. A coin generated from such a wallet will verify only when the same public key of the bank is used, and thus the merchant can check for himself the denomination of

| Program type | Prover (ms) | | Verifier (ms) | | Proof size | Cache size | Multi-exps | |
|---|---|---|---|---|---|---|---|---|
| | With cache | Without | With cache | Without | (bytes) | (Mbytes) | Prover | Verifier |
| DLR proof | 3.07 | 3.08 | 1.26 | 1.25 | 511 | 0 | 2 | 1 |
| Multiplication proof | 2.03 | 4.07 | 1.66 | 2.32 | 848 | 33.5 | 8 | 2 |
| Range proof | 36.36 | 74.52 | 21.63 | 31.54 | 5455 | 33.5 | 31 | 11 |
| CL recipient proof | 119.92 | 248.31 | 70.76 | 112.13 | 19189 | 134.2 | 104 | 39 |
| CL issuer proof | 7.29 | 7.38 | 1.73 | 1.73 | 1097 | 0 | 2 | 1 |
| CL possession proof | 125.89 | 253.17 | 78.19 | 117.67 | 19979 | 134.2 | 109 | 40 |
| Verifiable encryption | 416.09 | 617.61 | 121.87 | 162.77 | 24501 | 190.2 | 113 | 42 |
| Coin | 134.37 | 271.34 | 83.01 | 121.83 | 22526 | 223.7 | 122 | 45 |

Table 1: Time (in milliseconds) and size (in bytes) required for each of our proofs, averaged over twenty runs. Timings are considered from both the prover and verifier sides, as are the number of multi-exponentiations, and are considered both with and without caching for fixed-based exponentiations; the size of the cache is also measured (in megabytes). As we can see, using caching results on average in a $48\%$ speed improvement for the prover, and a $31\%$ improvement for the verifier.

the coin.

The program in Section 5.3 also reflects our decision to randomize the user's spending order rather than having them perform a range proof that the coin index was contained within the proper range. As the random spending order does not reveal how many coins are left in the wallet, the user's privacy is still protected even though the index is publicly available. Furthermore, because range proofs are slow and require a fair amount of space (see Table 1 for a reminder), this optimization resulted in coins that were $20\%$ smaller and $21\%$ faster to generate and verify.

Finally, endorsed e-cash requires a random value contributed by both the merchant and the user. Since e-coin transactions should be done over a secure channel, in practice we expect that SSL connections will be used between the user and the merchant. One useful feature of an SSL connection is that it already provides both parties with shared randomness, and thus this randomness can be used in our library to eliminate the need for a redundant message.

## 6.2 Buying and Bartering

Our library implements two efficient optimistic fair exchange protocols for use with e-cash. Belenkiy et al. [9] provide a *buy* protocol for exchanging a coin with a file, while Küpçü and Lysyanksaya [51] provide a *barter* protocol for exchanging two files or blocks. The two protocols serve different purposes (buy vs. barter) and so we have implemented both.

Two of the main usage scenarios of fair exchange protocols are e-commerce and peer-to-peer file sharing [9]. In e-commerce, one needs to employ a buy protocol to ensure that both the user and the merchant are protected; the user receives her item while the merchant receives his payment. In a peer-to-peer file sharing scenario, peers exchange files or blocks of files. In this setting, it is more beneficial to barter for the blocks than to buy them one at a time; for an exchange of $n$ blocks, buying all the blocks requires $O(n)$ verifiable escrow operations (which, as discussed in Section 5.4, are quite costly), whereas bartering for the blocks requires only one such operation, regardless of the number of blocks exchanged.

Although the solution might seem to be to barter all the time and never buy, Belenkiy et al. suggest that both protocols are useful in a peer-to-peer file sharing scenario. Peers who have nothing to offer but would still like to download can offer to buy the files, while peers who would like only to upload and have no interest in downloading can act as the merchant and earn e-cash. Due to the resource considerations mentioned above, however, bartering should always be used if possible.

Because peers do not always know beforehand if they want to buy or barter for a file, we have modified the buy protocol to match up with the barter protocol in the first two messages. This modification, as well as outlines of both the protocols, can be seen in Figure 3. We further modified both protocols to let them exchange multiple blocks at once, so that one block of the fair exchange protocol might correspond to multiple blocks of the underlying file.

We give an overview of each protocol below, with the optimizations we have added. We have also implemented the trusted third parties (the bank and the arbiter) necessary for e-cash and fair exchange. Although we do not describe in detail the resolution and bank interaction protocols, these can be found in the original papers [9, 51] and we provide performance benchmarks for the bank in Table 2.

### 6.2.1 Buying

The modified buy protocol is depicted on the left in Figure 3, although we also allow for the users to participate in the original buy protocol (in which the messages appear in a slightly different order). To initiate the modified buy protocol, the buyer sends a "setup" message, which consists of an unendorsed coin and a verifiable escrow on its corresponding endorsement. Upon receiving



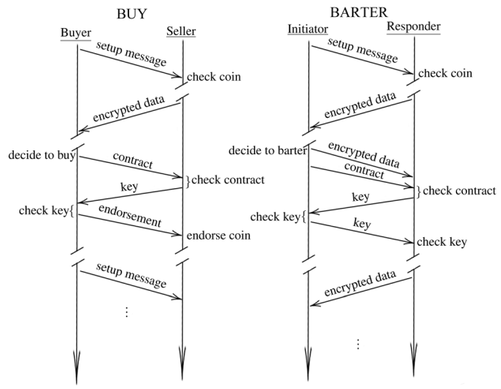



Figure 3: This figure provides outlines of both our buy and barter protocols [9, 51]. Until the decision to buy or barter, the two protocols are identical; the main difference is that in a buy protocol, the setup message must be sent for each file exchange, which results in a linear efficiency loss as compared to bartering.

this message, the seller will use the programs in Section 5 to check the validity of the coin and the escrow. If these proofs verify, the seller will proceed by sending back an encrypted version of his file (or file block). Upon receiving this ciphertext, the buyer will store it (and a Merkle hash of it, for use with the arbiter in case the protocol goes wrong later on) and send back a contract, which consists of a hash of the seller's file and some session information. The seller will check this contract and, if satisfied with the details of the agreement, send back its decryption key. The buyer can then use this key to decrypt the ciphertext it received in the second message of the protocol. If the decryption is successful, the buyer will send back his endorsement on the coin. If in these last steps either party is unsatisfied (for example, the file does not decrypt or the endorsement isn't valid for the coin from the setup message), they can proceed to contact the arbiter and run resolution protocols [9].

### 6.2.2 Bartering

This protocol is depicted on the right in Figure 3; because the first two messages of the barter protocol (the setup message and the encrypted data) are identical to those in the buy protocol described in the previous section, we do not describe them again here and instead jump directly to the third message. Because bartering involves an exchange of data, the initiator will respond to the receipt of the ciphertext with a ciphertext of her own, corresponding to an encryption of her file. She will also send a contract, which is similar to the buy contract but also contains hash information for her file. The responder will then check this contract as the seller did in the buy protocol, and if satisfied with the agreement will send back his decryption key. If the ciphertext decrypts correctly (i.e., decrypts to the file described in the contract) then the initiator can respond in turn with her own decryption key. If this decryption key is also valid, both parties have successfully obtained the desired files and the barter protocol can be considered complete. If neither party had to contact the arbiter (for similar reasons as in the buy protocol; i.e., a file did not decrypt correctly) then they are free to engage in future barter protocols without the overhead of an additional setup message. Otherwise, they need to resolve with the arbiter [51].

## 6.3 Library performance

In Table 2, we can see the computation time and size complexity for the steps described above, as well as computation and communication overhead for the withdraw and deposit protocols involving the bank. The numbers in the table were computed on the same computer as those in Section 5.4.

The numbers in Table 2 clearly demonstrate our earlier observation that bartering is considerably more efficient than buying, both in terms of computation and communication overhead. The setup message for both buying and bartering takes about $600$ms to generate and approximately $46$kB of space. In contrast, the rest of the barter protocol takes very little time; on the order of milliseconds for both parties (and about $1.5$kB of total overhead).

In addition, we consider the same protocols run using a previous "naïve" version of our library, which provided the same e-cash API and employed some multi-exponentation optimizations, but did not use ZKPDL. Using the optimizations available to the interpreter is considerably faster over our previous approach, meaning that our interpreter has not only made developing our protocols more convenient, but has also helped to improve efficiency.

# 7 Related work

Similar to our approach, FairPlayMP [13] (and its predecessor, FairPlay [57]) provides a language-based system for secure multi-party computation, allowing multiple parties to jointly compute a function on private inputs while revealing nothing but the resulting value. At the heart of FairPlayMP is a programming language, SFDL 2.0 (short for Secure Function Definition Language), that allows programmers to specify a multi-party computation. The authors provide a compiler that transforms SFDL programs into boolean circuits, and an engine that securely evaluates these circuits and distributes the resulting values among the involved parties. Although

| Operation | Time (ms) | "Naïve" time (ms) | Size (B) |
|---|---|---|---|
| Withdraw (user) | 126.35 | 290.79 | 20093 |
| Withdraw (bank) | 83.36 | 140.02 | 1167 |
| Deposit (bank) | 82.11 | 128.36 | 22526 |
| Buying a block (buyer) | 628.49 | 901.04 | 47286 |
| Buying a block (seller) | 211.89 | 275.94 | 203 |
| Barter setup message | 608.29 | 881.32 | 46934 |
| Checking setup message | 210.61 | 276.98 | n/a |
| Barter after setup (initiator) | 18.02 | 18.28 | 1280 |
| Barter after setup (responder) | 1.11 | 1.18 | 204 |

Table 2: Average time required and network overhead, in milliseconds and bytes respectively, for each stage in our e-cash implementation. The timings were averaged over twenty runs, and caching and compression optimizations were used. For the naïve timings, an older version of the library was used, which uses some multi-exponentation optimization techniques but not the interpreter; we can see a clear improvement when using ZKPDL. Parameters were used to provide a security level of 80 bits (160-bit SHA-1 hashing, 128-bit AES encryption, 1024-bit RSA moduli, and 1024-bit DSA signatures).

this is a very useful tool, it uses generic circuit techniques, and thus from an efficiency standpoint it is often desirable to instead develop a multi-party computation scheme specific to the intended application.

IBM's Idemix (identity mixer) project [19, 14] has independently developed a library for zero-knowledge proofs and anonymous credentials using Java. Idemix has focused on supporting the deployment of anonymous credentials in privacy-preserving identity systems, and provides a system for obtaining, proving, and verifying credentials using XML messages. The Idemix team has also developed a high-level language for zero-knowledge protocols, and describe a proof-of-concept compiler that can output Java or LaTeX code from these descriptions [5, 4]. However they do not provide performance benchmarks, and many implementation details are left as future work; neither the language nor the compiler appear in the released Idemix library. While Idemix and our work both provide implementations of anonymous credentials and CL signatures, in contrast, our focus on efficient, repeated execution of e-cash transactions has led us to pursue an alternate language-based strategy and develop a performance-optimized interpreter engine. We believe our runtime engine, ease of extensibility, and performance optimizations provide greater support to cryptographic researchers and systems programmers seeking a framework for deploying zero-knowledge protocols.

There are also compilers available [15, 6] for the generation of proofs of security and correctness for cryptographic protocols. While this is an interesting and important area of research, these tools largely focus on static analysis of protocols rather than performance. Perhaps more similar to our work, the languages Cryptol [53] and Stupid [52] provide a simple interface for developing low-level implementations of cryptographic primitives (such as hash functions) which can then be analyzed and translated into native code on different platforms.

## 8 Conclusions and Future Work

In this paper we have introduced a language for generating (and verifying) widely-used zero-knowledge proofs of knowledge. Through sample programs, we have demonstrated how our language is used to express advanced cryptographic primitives such as blind signatures, verifiable encryption, and endorsed e-cash. We presented optimizations provided by our language's interpreter and have shown they provide significant benefit.

Atop our language framework, we built a library that provides optimistic fair exchange protocols based on electronic cash. We have further presented optimizations for the protocols provided by Cashlib and argued for their practicality in network-based applications.

Much future work is possible for the ZKPDL language and interpreter. There are many other cryptographic primitives which could be incorporated into the language (e.g., encryption, signatures, hash functions), and other zero-knowledge protocols that could be added as relations (e.g., alternate and "fuzzy" schemes for range proofs). Incorporating these primitives, perhaps by allowing for subroutines and the composability of ZKPDL programs, would allow our library to be more easily extended and potentially have applicability to a broader range of secure systems. The analysis of ZKPDL programs—e.g., to automatically verify protocols and identify security errors through type analysis or formal verification techniques—provides another interesting area of study.

For increased performance on multicore architectures, we are working on analyzing dependencies among the expressions evaluated by our interpreter. The simplicity of our language, e.g., in `compute` blocks, allows a coarse-grained approach, as the only dependencies that arise between lines of ZKPDL are from variables which have been declared and assigned in previous lines.



Finally, in terms of extending Cashlib, to improve a bank's efficiency it might also be possible to speed up coin verification time by supporting batch verification techniques [22, 39] for CL signatures; we leave this as one of many interesting open problems.

## Acknowledgments

We gratefully acknowledge George Danezis, our shepherd, and our anonymous reviewers for their valuable feedback on earlier versions of this paper. We also would like to thank Gabriel Bender and Alex Hutter for their work developing earlier versions of Cashlib, Carleton Coffrin for assistance with ANTLR, and Jan Camenisch for his helpful discussions regarding the IBM Idemix Project. This work is supported in part by NSF CyberTrust grant 0627553.

## References

[1] Androulaki, E., Raykova, M., Srivatsan, S., Stavrou, A., and Bellovin, S. PAR: payment for anonymous routing. In *Privacy Enhancing Technologies Symposium (PETS)* (2008), vol. 5134 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 219–236.

[2] Asokan, N., Shoup, V., and Waidner, M. Optimistic fair exchange of digital signatures. In *Proc. Eurocrypt '98* (1998), vol. 1403 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 591–606.

[3] Avoine, G., and Vaudenay, S. Optimistic fair exchange based on publicly verifiable secret sharing. In *ACISP* (2004), vol. 3108 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 74–85.

[4] Bangerter, E., Barzan, S., Krenn, S., Sadeghi, A.-R., Schneider, T., and Tsay, J.-K. Bringing zero-knowledge proofs of knowledge to practice. In *17th International Workshop on Security Protocols* (2009).

[5] Bangerter, E., Camenisch, J., Krenn, S., Sadeghi, A.-R., and Schneider, T. Automatic generation of sound zero-knowledge protocols. Cryptology ePrint Archive, Report 2008/471, 2008. `http://eprint.iacr.org/2004/064`.

[6] Barbosa, M., Noad, R., Page, D., and Smart, N. First steps toward a cryptography-aware language and compiler. Cryptology ePrint Archive, Report 2005/160, 2005. `http://eprint.iacr.org/2005/160`.

[7] Belenkiy, M., Camenisch, J., Chase, M., Kohlweiss, M., Lysyanskaya, A., and Shacham, H. Delegatable anonymous credentials. In *Proc. Crypto '09* (2009), vol. 5677 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 108–125.

[8] Belenkiy, M., Chase, M., Erway, C., Jannotti, J., Küpçü, A., and Lysyanskaya, A. Incentivizing outsourced computation. In *NetEcon* (2008), pp. 85–90.

[9] Belenkiy, M., Chase, M., Erway, C., Jannotti, J., Küpçü, A., Lysyanskaya, A., and Rachlin, E. Making P2P accountable without losing privacy. In *WPES* (2007), ACM, pp. 31–40.

[10] Belenkiy, M., Chase, M., Kohlweiss, M., and Lysyanskaya, A. Non-interactive anonymous credentials. In *Proc. 5th Theory of Cryptography Conference (TCC)* (2008), pp. 356–374.

[11] Bellare, M., and Goldreich, O. On defining proofs of knowledge. In *Proc. Crypto '92* (1992), vol. 740 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 390–420.

[12] Bellare, M., and Rogaway, P. Random oracles are practical: a paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security (CCS) '93* (1993), pp. 62–73.

[13] Ben-David, A., Nisan, N., and Pinkas, B. FairplayMP: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (CCS) '08* (2008), pp. 257–266.

[14] Bichsel, P., Binding, C., Camenisch, J., Gross, T., Heydt-Benjamin, T., Sommer, D., and Zaverucha, G. Cryptographic protocols of the identity mixer library, v. 1.0. IBM Research Report RZ3730, 2009.

[15] Blanchet, B., and Pointcheval, D. Automated security proofs with sequences of games. In *Proc. Crypto '06* (2006), vol. 4117 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 537–554.

[16] Blum, M., de Santis, A., Micali, S., and Persiano, G. Non-interactive zero-knowledge. *SIAM Journal of Computing 20*, 6 (1991), 1084–1118.

[17] Boudot, F. Efficient proofs that a committed number lies in an interval. In *Proc. Eurocrypt '00* (2000), vol. 1807 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 431–444.

[18] Camenisch, J., Chaabouni, R., and abhi shelat. Efficient protocols for set membership and range proofs. In *Proc. Asiacrypt '08* (2008), pp. 234–252.

[19] Camenisch, J., and Herreweghen, E. V. Design and implementation of the idemix anonymous credential system. In *ACM Conference on Computer and Communications Security (CCS) '02* (2002), ACM, pp. 21–30.

[20] Camenisch, J., Hohenberger, S., Kohlweiss, M., Lysyanskaya, A., and Meyerovich, M. How to win the clonewars: efficient periodic n-times anonymous authentication. In *ACM Conference on Computer and Communications Security (CCS) '06* (2006), pp. 201–210.

[21] Camenisch, J., Hohenberger, S., and Lysyanskaya, A. Compact e-cash. In *Proc. Eurocrypt '05* (2005), vol. 3494 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 302–321.

[22] Camenisch, J., Hohenberger, S., and Pedersen, M. Ø. Batch verification of short signatures. In *Proc. Eurocrypt '07* (2007), vol. 4515 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 246–263.

[23] Camenisch, J., and Lysyanskaya, A. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Proc. Eurocrypt '01* (2001), vol. 2045 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 93–118.

[24] Camenisch, J., and Lysyanskaya, A. A signature scheme with efficient protocols. In *Proc. SCN '02* (2002), vol. 2576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 268–289.

[25] Camenisch, J., and Lysyanskaya, A. Signature schemes and anonymous credentials from bilinear maps. In *Proc. Crypto '04* (2004), vol. 3152 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 56–72.

[26] Camenisch, J., Lysyanskaya, A., and Meyerovich, M. Endorsed e-cash. In *IEEE Symposium on Security and Privacy* (2007), pp. 101–115.

[27] CAMENISCH, J., AND SHOUP, V. Practical verifiable encryption and decryption of discrete logarithms. In *Proc. Crypto '03* (2003), vol. 2729 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 126–144.

[28] CHAUM, D. Blind signatures for untraceable payments. In *Proc. Crypto '82* (1982), Lecture Notes in Computer Science, Springer-Verlag, pp. 199–203.

[29] CHAUM, D. Security without identification: transaction systems to make big brother obsolete. *Communications of the ACM 28*, 10 (1985), 1030–1044.

[30] CHAUM, D., DEN BOER, B., VAN HEYST, E., MJØLSNES, S. F., AND STEENBEEK, A. Efficient offline electronic checks (extended abstract). In *Proc. Eurocrypt '89* (1989), pp. 294–301.

[31] COX, B., TYGAR, J., AND SIRBU, M. Netbill security and transaction protocol. In *Proc. 1st Usenix Workshop on Electronic Commerce* (1995), pp. 77–88.

[32] DAEMEN, J., AND RIJMEN, V. *Rijndael: AES – The Advanced Encryption Standard*. Springer-Verlag, 2002.

[33] DAMGÅRD, I. On sigma protocols. `http://www.daimi.au.dk/ivan/Sigma.pdf`.

[34] DAMGÅRD, I. Payment systems and credential mechanism with provable security against abuse by individuals. In *Proc. Crypto '88* (1988), vol. 403 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 328–335.

[35] DAMGÅRD, I., DUPONT, K., AND PEDERSEN, M. Ø. Unclonable group identification. In *Proc. Eurocrypt '06* (2006), vol. 4004 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 555–572.

[36] DAMGÅRD, I., AND FUJISAKI, E. A statistically-hiding integer commitment scheme based on groups with hidden order. In *Proc. Asiacrypt '02* (2002), vol. 2501 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 125–142.

[37] DAMGÅRD, I., GROTH, J., AND SALOMONSEN, G. The theory and implementation of an electronic voting system. In *Proc. Secure Electronic Voting (SEC)* (2003), pp. 77–100.

[38] FEIGE, U., LAPIDOT, D., AND SHAMIR, A. Multiple non-interactive zero-knowledge proofs based on a single random string. In *Proc. 31st Symposium on Theory of Computing (STOC)* (1990), pp. 308–317.

[39] FERRARA, A. L., GREEN, M., HOHENBERGER, S., AND PEDERSEN, M. Ø. Practical short signature batch verification. In *Proc. CT-RSA* (2009), pp. 309–324.

[40] FIAT, A., AND SHAMIR, A. How to prove yourself: practical solutions to identification and signature problems. In *Proc. Crypto '86* (1986), vol. 263 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 186–194.

[41] FUJISAKI, E., AND OKAMOTO, T. Statistical zero knowledge protocols to prove modular polynomial relations. In *Proc. Crypto '97* (1997), vol. 1294 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 16–30.

[42] GENNARO, R., GENTRY, C., AND PARNO, B. Non-interactive verifiable computing: outsourcing computation to untrusted workers. Cryptology ePrint Archive, Report 2009/547, 2009. `http://eprint.iacr.org/2009/547`.

[43] GMP. The GNU MP Bignum library. `http://gmplib.org`.

[44] GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. How to construct random functions (extended abstract). In *Proc. 25th Symposium on the Foundations of Computer Science (FOCS)* (1984), pp. 464–479.

[45] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *J. ACM 38*, 3 (1991), 691–729.

[46] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof systems. In *Proc. 17th Symposium on the Theory of Computing (STOC)* (1985), pp. 186–208.

[47] GOLDWASSER, S., MICALI, S., AND RIVEST, R. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing 17*, 2 (1988), 281–308.

[48] GROTH, J. Non-interactive zero-knowledge arguments for voting. In *ACNS* (2005), vol. 3531 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 467–482.

[49] HEYDT-BENJAMIN, T., CHAE, H.-J., DEFEND, B., AND FU, K. Privacy for public transportation. In *Privacy Enhancing Technologies Symposium (PETS)* (2006), pp. 1–19.

[50] ISHIDA, N., MATSUO, S., AND OGATA, W. Divisible voting scheme. In *Proc. ISC '03* (2003), vol. 2851 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 137–150.

[51] KÜPÇÜ, A., AND LYSYANSKAYA, A. Usable optimistic fair exchange. In *Proc. CT-RSA '10* (2010), vol. 5985 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 252–267.

[52] LAURIE, B., AND CLIFFORD, B. Stupid: a metalanguage for cryptography, 2010. `http://code.google.com/p/stupid-crypto`.

[53] LEWIS, J., AND MARTIN, B. Cryptol: high assurance, retargetable crypto development, and validation. In *Proc. Military Communications Conference '03* (2003), pp. 820–825.

[54] LIPMAA, H. On Diophantine complexity and statistical zero-knowledge arguments. In *Proc. Asiacrypt '03* (2003), vol. 2894 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 398–415.

[55] LIPMAA, H., ASOKAN, N., AND NIEMI, V. Secure vickrey auctions without threshold trust. In *Proc. Financial Cryptography '02* (2002), vol. 2357 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 87–101.

[56] LYNN, B. PBC (pairing-based cryptography) library. `http://crypto.stanford.edu/pbc`.

[57] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay - a secure two-party computation system. In *USENIX Security Symposium* (2004), pp. 287–302.

[58] MENEZES, A. J., VAN OORSCHOT, P., AND VANSTONE, S. *Handbook of Applied Cryptography*. CRC Press, 1997.

[59] MERKLE, R. A digital signature based on a conventional encryption function. In *Proc. Crypto '88* (1987), vol. 293 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 369–378.

[60] NAOR, M., AND YUNG, M. Universal one-way hash functions and their cryptographic applications. In *Proc. 21st Symposium on Theory of Computing (STOC)* (1989), pp. 33–43.

[61] NGUYEN, L., AND SAFAVI-NAINI, R. Dynamic k-times anonymous authentication. In *ACNS* (2005), vol. 3531 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 318–333.

[62] PAGNIA, H., AND GÄRTNER, F. On the impossibility of fair exchange without a trusted third party. Darmstadt University Technical Report TUD-BS-1999-02, 1999.

[63] PARR, T. ANTLR parser generator. `http://www.antlr.org`.

[64] PEDERSEN, T. P. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proc. Crypto '91* (1992), vol. 576 of *Lecture Notes in Computer Science*, Springer-Verlag.

[65] RABIN, M., AND SHALLIT, J. Randomized algorithms in number theory. *Communications on Pure and Applied Mathematics 39*, 1 (1986), 239–256.

[66] SCHNORR, C.-P. Efficient signature generation by smart cards. *Journal of Cryptology 4*, 3 (1991), 161–174.

[67] SCHOENMAKERS, B. Interval proofs revisited. In *International Workshop on Frontiers in Electronic Elections* (2005).




# P4P: Practical Large-Scale Privacy-Preserving Distributed Computation Robust against Malicious Users


Yitao Duan
*NetEase Youdao*
*Beijing, China*
*duan@rd.netease.com*

John Canny
*Computer Science Division*
*University of California, Berkeley*
*jfc@cs.berkeley.edu*

Justin Zhan
*National Center for the Protection of Financial Infrastructure*
*South Dakota, USA*
*justin.zhan@ncpfi.org*



## Abstract

In this paper we introduce a framework for privacy-preserving distributed computation that is practical for many real-world applications. The framework is called Peers for Privacy (P4P) and features a novel heterogeneous architecture and a number of efficient tools for performing private computation and ensuring security at large scale. It maintains the following properties: (1) Provably strong privacy; (2) Adequate efficiency at reasonably large scale; and (3) Robustness against realistic adversaries. The framework gains its practicality by decomposing data mining algorithms into a sequence of vector addition steps that can be privately evaluated using a new verifiable secret sharing (VSS) scheme over *small* field (e.g., 32 or 64 bits), which has the same cost as regular, non-private arithmetic. This paradigm supports a large number of statistical learning algorithms including SVD, PCA, $k$-means, ID3, EM-based machine learning algorithms, etc., and all algorithms in the statistical query model [36]. As a concrete example, we show how singular value decomposition (SVD), which is an extremely useful algorithm and the core of many data mining tasks, can be done efficiently with privacy in P4P. Using real-world data and actual implementation we demonstrate that P4P is orders of magnitude faster than existing solutions.


## 1 Introduction

Imagine the scenario where a large group of users want to mine their collective data. This could be a community of movie fans extracting recommendations from their ratings, or a social network voting for their favorite members. In all the cases, the users may wish not to reveal their private data, not even to a "trusted" service provider, but still obtain verifiably accurate results. The major issues that make this kind of tasks challenging are the scale of the problem and the need to deal with cheating users. Typically the quality of the result increases with the size of the data (both the size of the user group and the dimensionality of per user data). Nowadays it is common for commercial service providers to run algorithms on data set collected from thousands or even millions of users. For example, the well-publicized Netflix Prize (http://www.netflixprize.com/) data set consists of roughly 100M ratings of 17,770 movies contributed by 480K users. At such a scale, both private computation and verifying proper behavior become impractical (more on this). In other words, privacy technologies fail to catch up with data mining algorithms's appetite and processing capability for large data sets.

We strive to change this. Our goal is to provide a privacy solution that is practical for many (but not all) real-world applications at reasonably large scale. We introduce a framework called Peers for Privacy (P4P) which is guided by the natural incentives of users/vendors and today's computing reality. On a typical computer today there is a six orders of magnitude difference between the crypto operations in large field needed for secure homomorphic computation (order of milliseconds) and regular arithmetic operations in small (32- or 64-bit) fields (fraction of a nano-second). Existing privacy solutions such as [11, 29] make heavy use of public-key operations for information hiding or verification. While they have the same asymptotic complexity as the standard algorithms for those problems, the constant factors imposed by public-key operations are prohibitive for large-scale systems. We show in section 3.3 and section 7.2 that they cannot be fixed with trivial changes to support applications at our scale. In contrast, P4P's main computation is based on verifiable secret sharing (VSS) over *small* field. This allows private arithmetic operations to have the *same* cost as regular, non-private arithmetic since both are manipulating the same-sized numbers with similar complexity. Moreover, such a paradigm admits extremely efficient zero-knowledge (ZK) tools that are practical even at large scale. Such tools are indispens-

able in dealing with cheating participants.

Some of techniques used in P4P were initially introduced in [21]. However, the focus of [21] is to develop an efficient zero-knowledge proof (ZKP) (for detecting cheating users) and prove its effectiveness. It leaves open how the ZKP should be incorporated into the computation to force proper behavior. As we will show, this is not trivial and requires additional tools, probably tailored to each application. In particular, [21] does not deal with the threat of cheating users changing their data during the computation. This could cause the computation to produce incorrect results. Such practical issues are not addressed in [21].

We fill in the missing pieces and provide a comprehensive solution. The contributions of this paper are: (1) We identify three key qualifications a practical privacy solution must possess, examine them in light of the changes in large-scale distributed computing, and formulate our design. The analysis not only provides rationales for our scheme, but also can serve as a guideline for practitioners to appraise the cost for obtaining privacy in their applications. (2) We introduce a new ZK protocol that verifies the consistency of user's data during the computation. This protocol complements the work of [21] and ensures the correctness of the computation in the presence of active user cheating. (3) We demonstrate the practicality of the framework with a concrete example, a private singular value decomposition (SVD) protocol. Prior to our work, there is no privacy solution providing comparable performance at such large scales. The example also serves as a tutorial showing how the framework can be adapted to different applications. (4) We have implemented the framework and performed evaluations against alternative privacy solutions on real-world data. Our experiments show a dramatic performance improvement. Furthermore, we have made the code freely available and are continuing to improve it. We believe that, like other secure computation implementations such as [46, 39, 5, 40], P4P is a very useful tool for developing privacy-preserving systems and represents a significant step towards making privacy a practical goal in real-world applications.

## 2 Preliminaries

We say that an adversary is passive, or semi-honest, if it tries to compute additional information about other player's data but still follows the protocol. An active, or malicious adversary, on the other hand, can deviate arbitrarily from the protocol, including inputting bogus data, producing incorrect computation, and aborting the protocol prematurely. Clearly active adversary is much more difficult to handle than passive ones. Our scheme is secure against a hybrid threat model that includes both passive and active adversaries. We introduce the model in section 4.

The privacy guarantee P4P provides is *differential privacy*, a notion of privacy introduced in [25], further refined by [24, 23], and adopted by many latest works such as [9, 43, 42, 8, 41]. Differential privacy models the leakage caused by releasing some function computed over a data set. It captures the intuition that the function is private if the risk to one's privacy does not substantially increase as a result of participating in the data set. Formally it is defined as:

**Definition 1 (Differential Privacy [25, 24])** $\forall \epsilon, \delta \geq 0$, *an algorithm $\mathcal{A}$ gives $(\epsilon, \delta)$-differential privacy if for all $S \subseteq Range(\mathcal{A})$, for all data sets $D, D'$ such that $D$ and $D'$ differ by a single record*

$$\Pr[\mathcal{A}(D) \in S] \leq \exp(\epsilon) \Pr[\mathcal{A}(D') \in S] + \delta$$

There are several solutions achieving differential privacy for some machine learning and data mining algorithms (e.g., [24, 9, 43, 42, 8, 41]). Most require a trusted server hosting the entire data set. Our scheme removes such a requirement and also provides tools for handling a more adversarial setting where the data sources may be malicious. [4] is also a distributed and differentially private scheme for binary sum functions but it is only secure in a semi-honest model.

Differential privacy is widely used in the database privacy community to model the leakage caused by answering queries. P4P's reliance on differential privacy is as follows: During the computation, certain aggregate information (including the final result) is released (other information is kept hidden using cryptographic means). This is also modeled as query responses computed over the entire data set. Measuring such leakage against differential privacy allows us to have a rigorous formulation of the risk each individual user faces. By tuning the parameters $\epsilon$ and $\delta$ we can control such risk and obtain a system with adequate privacy as well as high efficiency. Another nice property of using differential privacy is that it can cover the final results (in contrast secure MPC in cryptography does not) therefore the protection is complete. Integrating differential privacy into secure computation has been accepted by the cryptography community [4] and our work can been seen as a concrete and highly efficient instantiation of such an approach to secure computation of some algorithms.

## 3 Design Considerations

Our design was motivated by careful evaluation of goals, available resources, and alternative solutions.



### 3.1 Design Goals

Our goal is to provide practical privacy solutions for some real-world applications. To this end, we identify three properties that are essential to a practical privacy solution:

1. **Provable Privacy**: Its privacy must be rigorously proven against well formulated privacy definitions.

2. **Efficiency and Scalability**: It must have adequate efficiency at reasonably large scale, which is an absolute necessity for many of today's data mining applications. The scale we are targeting is unprecedented: to support real-world application both the number of users and the number of data items per user are assumed to be in millions.

3. **Robustness**: It must be secure against realistic adversaries. Many computations either involve the participation of users, or collect data from them. Cheating of a small number of users is a realistic threat that the system must handle.

To the best of our knowledge, no existing works, or trivial composition of them, attain all three. Ours is the first, with open-source code, supporting all these properties.

### 3.2 Available Resources

During the past few years the landscape of large-scale distributed computing has changed dramatically. Many new resources and paradigms are available at very low cost and many computations that were infeasible at large scale in the past are now running routinely. One notable trend is the rapid growth of "cloud computing", which refers to the model where clients purchase computing cycles and/or storage from a third-party provider over the Internet. Vendors are sharing their infrastructures and allowing general users access to gigantic computing capability. Industrial giants such as Microsoft, IBM, Yahoo!, and Google are all key players in the game. Some of the cloud services (e.g., Amazon's Elastic Compute Cloud, http://aws.amazon.com/ec2.) are already available to general public at very cheap price.

The growth of cloud computing symbolizes the increased availability of large-scale computing power. We believe it is time to re-think the issue of privacy preserving data mining in light of such changes. There are several significant differences:

1. Could computing providers have very different incentives. Unlike traditional e-commerce vendors who are naturally interested in users data (e.g., purchase history), the cloud computing providers's commodity (CPU cycles and disk space) is *orthogonal* to users' computation. Providers do not benefit directly from knowing the data or computation results, other than ensuring that they are correct.

2. The traditional image of client-server paradigm has changed. In particular, the users have much more control over the data and the computation. In fact in many cases the cloud servers will be running code written by the customers. This is to be contrasted with traditional e-commere where there is a tremendous power imbalance between the service provider, who possesses all the information and controls what computation to perform, and the client users.

3. The servers are now clusters of hundreds or even thousands of machines capable of handling huge amount of data. They are not bottlenecks anymore.

Discrepancy of incentives and power imbalance have been identified as two major obstacles for the adoption of privacy technology by researchers examining privacy issues from legal and economic perspectives [26, 1]. Interestingly, both are greatly mitigated with the dawn of cloud computing. While traditional e-commerce vendors are reluctant to adopt privacy technologies, cloud providers would happily comply with customers instructions regarding what computation to perform. And once a treasure for the traditional e-commerce vendors, user data is now almost a burden for the cloud computing providers: storing the data not only costs disk space, but also may entail certain liability such as hosting illegal information. Some cloud providers may even choose not to store the data. For example, with Amazon's EC2 service, user data only persists during the computation.

We believe that cloud computing offers an extremely valuable opportunity for developing a new paradigm of practical privacy-preserving distributed computation: the existence of highly available, highly reputable, legally bounded service providers also provides a very important source of security. In particular, they make it realistic to treat some participants as *passive* adversaries. (The rests are still handled as active adversaries. The model is therefore a heterogenous one.) By tapping into this resource, we can build a heterogeneous system that can have privacy, scalability and robustness all at once.

### 3.3 The Alternatives

Existing privacy solutions for distributed data mining can be classified into two models: distributed and server-based. The former is represented by a large amount of work in the area of secure multiparty computation (MPC) in cryptography. The latter includes mostly homomorphic encryption-based schemes such as [11, 22, 51].

**Generic MPC**: MPC allows $n$ players to compute a function over their collective data without compromising the privacy of their inputs or the correctness of the outputs even when some players are corrupted by the same adversary. The problem dates back to Yao [52] and Goldreich et al. [31], and has been extensively studied in cryptography [6, 2, 33]. Recent years see some significant improvement in efficiency. Some protocols achieve nearly optimal asymptotic complexity [3, 16] while some work in small field [12].

From practitioners' perspective, however, these generic MPC protocols are mostly of theoretical interest. Reducing asymptotic complexity does not automatically make the schemes practical. These schemes tend to be complex which imposes a huge barrier for developers not familiar with this area. Trying to support generic computation, most of them compile an algorithm into a (boolean or arithmetic) circuit. Not only the depth of such a circuit can be huge for complex algorithms, it is also very difficult, if not entirely impossible, to incorporate existing infrastructures and tools (e.g., ARPACK, LAPACK, MapReduce, etc.), into such computation. These tools are indispensable part of our daily computing life and symbolize the work of many talents over many years. Re-building production-ready implementations is costly and error-prone and generally not an option for most companies in our fast-pacing modern world.

Recently there are several systems that implemented some of the MPC protocols. While this reflects a plausible attempt to bridge the gap between theory and practice, unfortunately, performance-wise none of the systems came close to providing satisfactory solutions for most large-scale real-world applications. Table 1 shows some representative benchmarks obtained by these implementations. Using FairplayMP [5] as an example, adding two 64-bit integers is compiled into a circuit of 628 gates and 756 wires using its SFDL compiler. According to [5]'s benchmark, evaluating such a circuit between two players takes about 7 seconds. With this performance, adding $10^6$ vectors of dimensionality $10^6$ each, which constitutes one iteration in our framework, takes $7 \times 10^{12}$ seconds, or 221,969 years.

**ECC and a single server**: It has been shown that conventional client-server paradigm can be augmented with homomorphic encryption to perform some computations with privacy (e.g., [11, 22, 51]). Still, such schemes are only marginally feasible for small to medium scale problems due to the need to perform at least linear number of large field operations even in purely semi-honest model. Using elliptic curve cryptography (ECC) can mitigate the problem as ECC can reduce the size of the cryptographic field (e.g., a 160-bit ECC key provides the same level of security as a 1024-bit RSA key). ECC cryptosystems such as [44] are $(+,+)$-homomorphic which is ideal for private computation. However, ECC point addition requires 1 field inversion and several field multiplications. The operation is still orders of magnitude slower than adding 64-bit or 32-bit integers directly. According to our benchmark, inversion and multiplication in a 160-bit field take 0.0224 and 0.001 milliseconds, respectively. Adding 1 million $10^6$-element vectors takes 260 days.

**Lesson learned**: For large-scale problems, privacy and security must be added with negligible cost. In particular, those steps that dominate the computation should *not* be burdened with public-key cryptographic operations (even those "efficient" ones such as ECC) simply because they have to be performed so many times. This is the major principle that guides our design. In our scheme, the main computation is always performed in small field, while verifications are done via random projection techniques to reduce the number of cryptographic operations. As our experiments show, this approach is effective. When the number of cryptographic operations are insignificant, even using the traditional ElGamal encryption (or commitment) with 1024-bit key the performance is adequate for large scale problems.

## 4 P4P's Architecture

Our approach is called Peers for Privacy, or P4P. The name comes from the feature that, during the computation, certain *aggregate* information is released. This is a very important technique that allows the private protocol to have high efficiency. We show that publishing such aggregate information does not harm privacy: individual traits are masked out in the aggregates and releasing them is safe. In other words, peers data mutually protects each other within the aggregates.

Let $\kappa > 1$ be a small integer. We assume that there are $\kappa$ servers belonging to different service providers (e.g., Amazon's EC2 service and Microsoft's Azure Services Platform, http://www.microsoft.com/azure/default.mspx). We define a *server* as all the computation units under the control of a single entity. It can be a cluster of thousands of machines so that it has the capability to support a large number of users.

**Threat Model** *Let $\alpha \in [0, 0.5)$ be the upper bound on the fraction of the dishonest users in the system.* [1] *Our scheme is robust against a computationally bounded adversary whose capability of corrupting parties is modeled as follows:*

1. *The adversary may actively corrupt at most $\lfloor \alpha n \rfloor$ users where $n$ is the number of users.*
2. *In addition to 1, we also allow the same adversary to passively corrupt $\kappa - 1$ server(s).*



Table 1: Performance Comparison of Existing MPC Implementations

| System | Adversary Model | Benchmark | Run Time (sec) |
|---|---|---|---|
| Fairplay [40] | Semi-honest | Billionaires | 1.25 |
| FairplayMP [5] | Semi-honest | Binary Tree Circuit (512 Gates) | 6.25 |
| PSSW [46] | Semi-honest | AES Encryption of 128-bit block | 7 |
| LPS [39] | Malicious | 16-bit Integer Comparison | 135 |

This model was proposed in [21] and is a special case of the general adversary structure introduced in [28, 34, 35] in that some of the participants are actively corrupted while some others are passively corrupted by the same adversary *at the same time*. Our model does not satisfy the feasibility requirements of [34, 35] and [28]. We avoid the impossibility by considering addition only computation.

The model models realistic threats in our target applications. In general, users are not trustworthy. Some may be incentivized to bias the computation, some may have their machines corrupted. So we model them as active adversaries and our protocol ensures that active cheating of a small number of users will not exert large influence on the computation. This greatly improves over existing privacy-preserving data mining solutions (e.g. [38, 51, 49]) and many current MPC implementations which handle only purely passive adversary. The servers, on the other hand, are selling CPU cycles and disk space, something that is not related to user's computation or data. Deviating from the protocol causes them penalty (e.g., loss of revenue for incorrect results) but little benefit. Their threat is therefore passive. (Corrupted servers are allowed to share data with corrupted users)

Treating "large institutional" servers as semi-honest, non-colluding has already been established by various previous work [38, 51, 50, 49]. However, in most of the models, the servers are not only semi-honest, but also "trusted", in that some user data is exposed to at least one of the servers (vertical or horizontal partitioned database). Our model does not have this type of trust requirement as each server only holds a random share of the user data. This further reduces the server's incentive to try to benefit from user data (e.g., reselling it) because the information it has are just random numbers without the other shares. A compromise requires the collusion of *all* servers which is a much more difficult endeavor. This also works for the servers' benefit: they are relieved of the liability of hosting secret or illegal computation, a problem that someone [18] envisions cloud providers will be facing.

## 5 The P4P Framework

Let $n$ be the number of users. Let $\phi$ be a small (e.g., 32- or 64-bit) integer. We write $\mathbb{Z}_\phi$ for the additive group of integers modulo $\phi$. Let $a_i$ be private user data for user $i$ and $I$ be public information. Both can be matrices of arbitrary dimensions with elements from arbitrary domains. Our scheme supports any iterative algorithms whose $(t+1)$-th update can be expressed as

$$I^{(t+1)} = f(\sum_{i=1}^{n} d_i^{(t)}, I^{(t)})$$

where $d_i^{(t)} = g(a_i, I^{(t)}) \in \mathbb{Z}_\phi^m$ is an $m$-dimensional data vector for user $i$ computed locally. Typical values for both $m$ and $n$ can range from thousands to millions. Both $f$ and $g$ are in general non-linear. In the SVD example that we will present, $I^{(t)}$ is the vector returned by ARPACK, $g$ is matrix-vector product, and $f$ is the internal computation performed by ARPACK.

This simple primitive is a surprisingly powerful model supporting a large number of popular data mining and machine learning algorithms, including Linear Regression, Naive Bayes, PCA, $k$-means, ID3, and EM etc., as has been demonstrated by numerous previous work such as [11, 13, 17, 10, 22]. It has been shown that all algorithms in the statistical query model [36] can be expressed in this form. Moreover, addition is extremely easy to parallelize so aggregating a large amount of numbers on a cluster is straightforward.

### 5.1 Private Computation

In the following we only describe the protocol for one iteration since the entire algorithm is simply a sequential invocations of the same protocol. The superscript is thus dropped from the notation. For simplicity, we only describe the protocol for the case of $\kappa = 2$. It is straightforward to extend it to support $\kappa > 2$ servers (by substituting the $(2, 2)$-threshold secret sharing scheme with a $(\kappa, \kappa)$ one). Using more servers strengthens the privacy protection but also incurs additional cost. We do not expect the scheme will be used with a large number of servers. This arrangement simplifies matters such as synchronization and agreement. Let $S_1$ and $S_2$ denote

the two servers. Leaving out validity and consistency check which will be illustrated using the SVD example, the basic computation is carried out as follows:

1. User $i$ generates a uniformly random vector $u_i \in \mathbb{Z}_\phi^m$ and computes $v_i = d_i - u_i \mod \phi$. She sends $u_i$ to $S_1$ and $v_i$ to $S_2$.
2. $S_1$ computes $\mu = \sum_{i=1}^n u_i \mod \phi$ and $S_2$ computes $\nu = \sum_{i=1}^n v_i \mod \phi$. $S_2$ sends $\nu$ to $S_1$.
3. $S_1$ updates $I$ with $f((\mu + \nu) \mod \phi, I)$.

It is straightforward to verify that if both servers follow the protocol, then the final result is indeed the sum of the user data vectors mod $\phi$. This result will be correct if every user's vector lies in the specified bounds for L2-norm, which is checked by the ZKP in [21].

## 5.2 Provable Privacy

**Theorem 1** *P4P's computation protocol leaks no information beyond the intermediate and final aggregates, if no more than $\kappa - 1$ servers are corrupted.*

The proof follows easily the fact that both the secret sharing scheme (for the computation) and the Pedersen commitment scheme [45, 15] used in the ZK protocols are information-theoretic private, as the adversary's view of the protocol is uniformly random and contains no information about user data. We refer the readers to [30] for details and formal definition of information-theoretic privacy.

As for the leakage caused by the released sums, first, for SVD, and some other algorithms, we are able to show the sums can be approximated from the final result so they do not leak more information. For general computation, we draw on the works on differential privacy. [20] has shown that, using well-established results from statistical database privacy [7, 19, 25], under certain conditions, releasing the vector sums still maintains differential privacy.

In some situations verifying the conditions of [20] privately is non-trivial but this difficulty is not essential in our scheme. There are well-established results that prove that differential privacy, as well as adequate accuracy, can be maintained as long as the sums are perturbed by independent noise with variance calibrated to the number of iterations and the sensitivity of the function [7, 19, 25]. In our settings, it is trivial to introduce noise into our framework – each server, which is semi-honest, can add the appropriate amount of noise to their partial sums after all the vectors from users are aggregated. Calibrating noise level is also easy: All one needs are the parameters $\epsilon, \delta$, the total number of queries ($mT$ in our case where $T$ is the number of iterations), and the sensitivity of the function $f$, which is summation in our case, defined as [25]:

$$S(f) = \max_{D,D'} \|f(D) - f(D')\|_1$$

where $D$ and $D'$ are two data sets differing by a single record and $\|\cdot\|_1$ denotes the L1-norm of a vector. Cauchy's Inequality states that

$$(\sum_{i=1}^m x_i y_i)^2 \le (\sum_{i=1}^m x_i^2)(\sum_{i=1}^m y_i^2)$$

For a user vector $a = [a_1, \ldots, a_m]$, let $x_i = |a_i|$, $y_i = 1$, we have

$$\|a\|_1^2 = (\sum_{i=1}^m |a_i|)^2 \le (\sum_{i=1}^m a_i^2)m = \|a\|_2^2 m$$

Since our framework bounds the L2-norm of a user's vector to below $L$, this means the sensitivity of the computation is at most $\sqrt{m}TL$.

Note that the perturbation does not interfere with our ZK verification protocols in any way, as the latter is performed between each user and the servers on the *original* data. Whether noise is necessary or not is dependent on the algorithm. For simplicity we will not describe the noise process in our protocol explicitly. We stress again that the SVD example we will present next does *not* need any noise at all. See section 6.6.

# 6 Private Large-Scale SVD

In the following we use a concrete example, a private SVD scheme, to demonstrate how the P4P framework can be used to support private computation of popular algorithms.

## 6.1 Basics

Recall that for a matrix $A \in \mathbb{R}^{n \times m}$, there exists a factorization of the form

$$A = U\Sigma V^T \quad (1)$$

where $U$ and $V$ are $n \times n$ and $m \times m$, respectively, and both have orthonormal columns. $\Sigma$ is $n \times m$ with nonnegative real numbers on the diagonal sorted in descending order and zeros off the diagonal. Such a factorization is called a singular value decomposition of $A$. The diagonal entries of $\Sigma$ are called the the singular values of $A$. The columns of $U$ and $V$ are left- resp. right-singular vectors for the corresponding singular values.

SVD is a very powerful technique that forms the core of many data mining and machine learning algorithms. Let $r = rank(A)$ and $u_i, v_i$ be the column vectors of $U$ and $V$, respectively. Equation 1 can be rewritten as $A = U\Sigma V^T = \sum_{i=1}^r \sigma_i u_i v_i^T$ where $\sigma_i$ is the $i$th singular value of $A$. Let $k \le r$ be an integer parameter, we can approximate $A$ by $A_k = U_k \Sigma_k V_k^T = \sum_{i=1}^k \sigma_i u_i v_i^T$. It is known that of all rank-$k$ approximations, $A_k$ is optimal in Frobenius norm sense. The $k$ columns of $U_k$ (resp. $V_k$) give the optimal $k$-dimensional approximation to the columnspace (resp. rowspace) of $A$. This dimensionality reduction preserves the structure of original data while considers only essential components of the matrix. It usually filters out noise and improves the performance of data mining tasks.

Our implementation uses a popular eigensolver, ARPACK [37] (ARnoldi PACKage), and its parallel version PARPACK. ARPACK consists of a collection of Fortran77 subroutines for solving large-scale eigenvalue problems. The package implements the Implicitly Restarted Arnoldi Method (IRAM) and allows one to compute a few, say $k$, eigenvalues and eigenvectors with user specified features such as those of largest magnitude. Its storage complexity is $nO(k) + O(k^2)$ where $n$ is the size of the matrix. ARPACK is a freely-available yet powerful tool. It is best suited for applications whose matrices are either sparse or not explicitly available: it only requires the user code to perform some "action" on a vector, supplied by the solver, at every IRAM iteration. This action is simply matrix-vector product in our case. Such a reverse communication interface works seamlessly with P4P's aggregation protocol.

## 6.2 The Private SVD Scheme

In our setting the rows of $A$ are distributed across all users. We use $A_{i*} \in \mathbb{R}^m$ to denote the $m$-dimensional row vector owned by user $i$. From equation 1, and the fact that both $U$ and $V$ are orthonormal, it is clear that $A^T A = V\Sigma^2 V^T$ which implies that $A^T A V = V\Sigma^2$. A straightforward way is then to compute $A^T A = \sum_{i=1}^n A_{i*}^T A_{i*}$ and solve for the eigenpairs of $A^T A$. The aggregate can be computed using our private vector addition framework. This is a distributed version of the method proposed in [7] and does not require the consistency protocol that we will introduce later. Unfortunately, this approach is not scalable as the cost for each user is $O(m^2)$. Suppose $m = 10^6$, and each element is a 64-bit integer, then $A_{i*}^T A_{i*}$ is $8 \times 10^{12}$ bytes, or about 8 TB. The communication cost for each user is then 16 TB (she must send shares to two servers). This is a huge overhead, both communication- and computation-wise. Usually the data is very sparse and it is a common practice to reduce cost by utilizing the sparsity. Unfortunately, sparsity does not help in a privacy-respecting application: revealing which elements are non-zero is a huge privacy breach and the users are forced to use the dense format. We propose the following scheme which

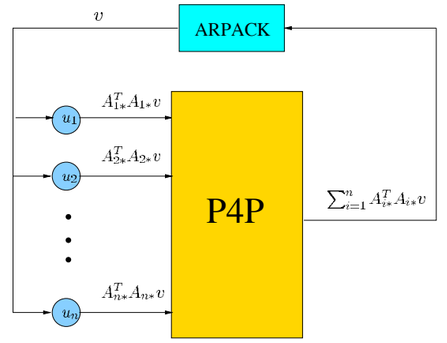



Figure 1: Private SVD with P4P

reduces the cost dramatically. We involve the users in the iteration and the total communication (and computation) cost per iteration is only $O(m)$ for each user. The number of iterations required ranges from tens to over a thousand. This translates to a maximum of a few GB data communicated for each user for the *entire* protocol which is much more manageable.

One server, say $S_1$, will host an ARPACK engine and interact with its reverse communication interface. In our case, since $A^T A$ is symmetric, the server will use `dsaupd`, ARPACK's double precision routine for symmetric problems, and asks for $k$ largest (in magnitude) eigenvalues. At each iteration, `dsaupd` returns a vector $v$ to the server code and asks for the matrix-vector product $A^T A v$. Notice that

$$A^T A v = \sum_{i=1}^n A_{i*}^T A_{i*} v$$

Each term in the summation is computable by each user locally in $O(m)$ time (by computing the inner product $A_{i*} \cdot v$ first) and the result is an $m$-vector. The vector can then be input to the P4P computation which aggregates them across all users privately. The aggregate is the matrix-vector product which can be returned to ARPACK for another iteration. This process is illustrated in figure 1.

The above method is known to have sensitivity problem, i.e., a small perturbation to the input could cause large error in the output. In particular, the error is $O(\|A\|^2/\sigma_k)$ [48]. Fortunately, most applications (e.g., PCA) only need the $k$ largest singular values (and their singular vectors). It is usually not a problem for those applications since for the principal components $O(\|A\|^2/\sigma_k)$ is small. There is no noticeable inaccuracy in our test applications (latent semantic analysis for document retrieval). For general problems the stable way is

to compute the eigenpairs of the matrix

$$H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$$

It is straightforward to adopt our private vector addition framework to compute matrix-vector product with $H$. For simplicity we will not elaborate on this.

## 6.3 Enforcing Data Consistency

During the iteration, user $i$ should input $d_i = A_{i*}^T A_{i*} v$. However, a cheating user could input something completely different. This threat is different from inputting bogus (but in the allowable range) data at the beginning (and using it consistently throughout the iterations). The latter only introduces noise to the computation but generally does not affect the convergence. The L2-norm ZKP introduced in [21], which verifies that the L-2 norm of a user's vector is bounded by a public constant, is effective in bounding the noise but does not help in enforcing consistency. The former, on the other hand, may cause the computation not to converge at all. This generally is a problem for iterative algorithms and is more than simply testing the equality of vectors: The task is complicated by the local function that each user uses to evaluate on her data, i.e., she is not simply inputting her private data vector, but some (possibly non-linear) function of it. In the case of SVD, the system needs to ensure that user $i$ uses the same $A_{i*}$ (to compute $d_i = A_{i*}^T A_{i*} v$) in all the iterations, not that she inputs the same vector.

We provide a novel zero-knowledge tool that ensures that the correct data is used. The protocol is probabilistic and relies on random projection. That is, the user is asked to project her original vector and her result of the current round onto some random direction. It then tests the relation of the two projections. We will show that this method catches cheating with high probability but only involves very few expensive large field operations.

### 6.3.1 Tools

The consistency protocol uses some standard cryptographic primitives. Detailed construction and proofs can be found in [45, 15, 11]. We summarize only their key properties here. All values used in these primitives lie in the multiplicative group $\mathbb{Z}_q^*$, or in the additive group of exponents for this group, where $q$ is a 1024 or 2048-bit prime. They rely on RSA or discrete log functions for cryptographic protection of information.

- **Homomorphic commitment**: A homomorphic commitment to an integer $a$ with randomness $r$ is written as $\mathcal{C}(a, r)$. It is homomorphic in the sense that $\mathcal{C}(a, r)\mathcal{C}(b, s) = \mathcal{C}(a+b, r+s)$. It is infeasible to determine $a$ given $\mathcal{C}(a, r)$. We say that a prover "opens" the commitment if it reveals $a$ and $r$.
- **ZKP of knowledge**: A prover who knows $a$ and $r$ (i.e., who knows how to open $\mathcal{A} = \mathcal{C}(a, r)$) can demonstrate that it has this knowledge to a verifier who knows only the commitment $\mathcal{A}$. The proof reveals nothing about $a$ or $r$.
- **ZKP for equivalence**: Let $\mathcal{A} = \mathcal{C}(a, r)$ and $\mathcal{B} = \mathcal{C}(a, s)$ be two commitments to the same value $a$. A prover who knows how to open $\mathcal{A}$ and $\mathcal{B}$ can demonstrate to a verifier in zero knowledge that they commit to the same value.
- **ZKP for product**: Let $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ be commitments to $a$, $b$, $c$ respectively, where $c = ab$. A prover who knows how to open $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ can prove in zero knowledge to a verifier who has only the commitments that the relationship $c = ab$ holds among the values they commit to. If say $a$ is made public, this primitive can be used to prove that $\mathcal{C}$ encodes a number that is multiple of $a$.

### 6.3.2 The Protocol

The consistency check protocol is summarized in the following. Since the protocol is identical for all users, we drop the user subscript for the rest of the paper whenever there is no confusion. Let $a \in \mathbb{Z}_\phi^m$ be a user's original vector (i.e., her row in the matrix $A$). The correct user input to this round should be $d = a^T av$. For two vectors $x$ and $y$, we use $x \cdot y$ to denote their scalar product.

1. After the user inputs her vector $d$, in the form of two random vectors $d^{(1)}$ and $d^{(2)}$ in $\mathbb{Z}_\phi^m$, one to each server, s.t. $d = d^{(1)} + d^{(2)} \mod \phi$, $S_1$ broadcasts a random number $r$. Using $r$ as the seed and a public PRG (pseudo-random generator), all players generate a random vector $c \in_R \mathbb{Z}_\phi^m$.
2. For $j \in \{1, 2\}$, the user computes $x^{(j)} = c \cdot a^{(j)} \mod \phi$, $y^{(j)} = a^{(j)} \cdot v \mod \phi$. Let $x = x^{(1)} + x^{(2)}$, $y = y^{(1)} + y^{(2)}$, $z = xy$. Let $w = (c \cdot a)(a \cdot v) - xy$. The user commits $\mathcal{X}^{(j)}$ to $x^{(j)}$, $\mathcal{Y}^{(j)}$ to $y^{(j)}$, $\mathcal{Z}$ to $z$, and $\mathcal{W}$ to $w$. She also construct two ZKPs: (1) $\mathcal{W}$ encodes a number that is multiple of $\phi$. (2) $\mathcal{Z}$ encodes a number that is the product of the two numbers encoded in $\mathcal{X}$ and $\mathcal{Y}$ where $\mathcal{X} = \mathcal{X}^{(1)}\mathcal{X}^{(2)}$ and $\mathcal{Y} = \mathcal{Y}^{(1)}\mathcal{Y}^{(2)}$. She sends all commitments and ZKPs to both servers.
3. The user opens $\mathcal{X}^{(j)}$ and $\mathcal{Y}^{(j)}$ to $S_j$ who verifies that both are computed correctly. Both servers verify the ZKPs. If any of them fails, the user is marked as FAIL and the servers terminate the protocol with her.
4. For $j \in \{1, 2\}$, the user computes $\tilde{z}^{(j)} = c \cdot d^{(j)} \mod \phi$, $\tilde{z} = \tilde{z}^{(1)} + \tilde{z}^{(2)}$ and $\tilde{w} = c \cdot d - \tilde{z}$. She commits $\tilde{\mathcal{Z}}^{(1)}$ to $\tilde{z}^{(1)}$, $\tilde{\mathcal{Z}}^{(2)}$ to $\tilde{z}^{(2)}$, and $\tilde{\mathcal{W}}$ to $\tilde{w}$. She constructs the following two ZKPs: (1) $\tilde{\mathcal{W}}$ encodes a number that is multiple of $\phi$ and (2)$\tilde{\mathcal{Z}}\tilde{\mathcal{W}}$ and $\mathcal{Z}\mathcal{W}$ encode the same value. She sends all the commitments and ZKPs to both servers.
5. The user opens $\tilde{\mathcal{Z}}^{(j)}$ to $S_j$ who verifies that it is computed correctly. Both servers verify the two ZKPs. They mark the user as FAIL if any of the verifications fails and terminate the protocol with her.
6. Both servers output PASS.

**Group Sizes**

There are three groups/fields involved in the protocol: the large, multiplicative group $\mathbb{Z}_q^*$ used for commitments and ZKPs, the "small" group $\mathbb{Z}_\phi$ used for additive secret-sharing, and the group of all integers. All the commitments such as $\mathcal{X}^{(j)}$ and $\mathcal{Y}^{(j)}$ are computed in $\mathbb{Z}_q^*$ so standard cryptographic tools can be used. The inputs to the commitments, which can be user's data or some intermediate results, are either in $\mathbb{Z}_\phi$ or in the integer group (without bounding their values). Restricting commitment inputs to small field/group does not compromise the security of the scheme since the outputs are still in the large field. Using Pederson's commitment as an example, the hiding property is guaranteed by the random numbers that are generated in the *large* field for each commitments. And breaking the binding property is still equivalent to solving the discrete logarithm problem in $\mathbb{Z}_q^*$. See [45].

The protocol makes it explicit which group a number is in using the $\mod \phi$ operator (i.e., $x = g(y) \mod \phi$ restricts $x$ to be in $\mathbb{Z}_\phi$ while $x = g(y)$ means $x$ can be in the whole integer range). The protocol assumes that $q \gg \phi$. This ensures that the numbers that are in the integer group ($x, y, z, w$ in step 2 and $\tilde{z}$ and $\tilde{w}$ in step 4) are much less than $q$ to avoid modular reduction when their commitments are produced. This is true for most realistic deployment, since $\phi$ is typically 64 bits or less while $q$ is 1024 bits or more. Theorem 2 proves that the transition from $\mathbb{Z}_\phi$ to integer fields and $\mathbb{Z}_q^*$ only causes the protocol to fail with extremely low probability:

**Theorem 2** *Let $O$ be the output of the Consistency Check protocol. Then*

$$\Pr(O = \textit{PASS} | d = a^T av) = 1$$

*and*

$$\Pr(O = \textit{PASS} | d \neq a^T av) \leq \frac{1}{\phi}$$

*Furthermore, the protocol is zero-knowledge.*

**Proof** If computed correctly, both $w$ and $\tilde{w}$ are multiples of $\phi$ due to modular reduction. Because of homomorphism, the equivalence ZKP that $\tilde{\mathcal{Z}}\tilde{\mathcal{W}}$ and $\mathcal{Z}\mathcal{W}$ encode the same value is to verify that $c \cdot d = c \cdot (a^T av)$.

Completeness: If the user performs the computation correctly, she should input $d = a^T av$ into this round of computation. All the verifications should pass. The protocol outputs PASS with probability 1.

Soundness: Suppose $d \neq aa^T v$. The user is forced to compute the commitments $\mathcal{X}^{(1)}, \mathcal{X}^{(2)}, \mathcal{Y}^{(1)}, \mathcal{Y}^{(2)}$, and $\tilde{\mathcal{Z}}^{(1)}, \tilde{\mathcal{Z}}^{(2)}$ faithfully since she has to open them to at least to one of the servers. The product ZKP at step 2 forces the number encoded in $\mathcal{Z}$ to be $xy$ which differs from $c \cdot (a^T av)$ by $w$. Due to homomorphism, at step 4, $\tilde{\mathcal{Z}}$ encodes a number that differs from $c \cdot d$ by $\tilde{w}$. The user could cheat by lying about $w$ or $\tilde{w}$, i.e., she could encode some other values in $\mathcal{W}$ and $\tilde{\mathcal{W}}$ to adjust for the difference between $c \cdot d$ and $c \cdot (a^T av)$, hoping to pass the equivalence ZKP. However, assuming the soundness of the ZKPs used, the protocol forces both to be multiple of $\phi$ (steps 2 and 4), so she could succeed only when the difference between $c \cdot d$, which she actually inputs to this round, and $c \cdot (a^T av)$, which she should input, is some multiple of $\phi$. Since $c$ is made known to her *after* she inputs $d$, the two numbers are totally unpredictable and random to her. The probability that $c \cdot d - c \cdot (a^T av)$ is a multiple of $\phi$ is only $1/\phi$ which is the probability of her success.

Finally, the protocol consists of a sequential invocation of some well-established ZKPs. By the sequential composition theorem of [32], the whole protocol is also zero-knowledge.

As a side note, all the ZKPs can be made non-interactive using the Fiat-Shamir paradigm [27]. The user could upload her data in a batch without further interaction. This makes it easier to deploy the scheme. It is also much more light-weight than the L2-norm ZKP [21]: the number of large field operations is *constant*, as opposed to $O(\log m)$ in the L2-norm ZKP. The private SVD computation thus involves only one L2-norm ZKP at first round, and one light verification for each of the subsequent rounds.

## 6.4 Dealing with Real Numbers

In their simplest forms, the cryptographic tools only support computation on integers. In most domains, however, applications typically have to handle real numbers. In the case of SVD, even if the original input matrix contains only integer entries, it is likely that real numbers appear in the intermediate (e.g., the vectors returned by ARPACK) and the final results.

Because of the linearity of the P4P computation, we can use a simple linear digitization scheme to convert

between real numbers in the application domain and $Z_\phi$, P4P's integer field. Let $R > 0$ be the bound of the maximum absolute value application data can take, i.e., all numbers produced by the application are between $[-R, R]$. The integer field provides $|\phi|$ bits resolution. This means the maximum quantization error for one variable is $R/\phi = 2^{|R|-|\phi|}$. Summing across all $n$ users, the worst case absolute error is bounded by $n2^{|R|-|\phi|}$. In practice $|\phi|$ can be 64, and $|R|$ can be around e.g., 20 (this gives a range of $[-2^{20}, 2^{20}]$). With $n = 10^6$, this gives a maximum absolute error of under 1 over a million.

### 6.5 The Protocol

Let $Q$ be the set of qualified users initialized to the set of all users. The entire private SVD method is summarized as follows:

1. **Input** The user first provides an L2-norm ZKP [21] on $a$ with a bound $L$, i.e., she submits a ZKP that $\|a\|_2 < L$. This step also forces the user to commit to the vector $a$. Specifically, at the end of this step, $S_1$ and $S_2$ have $a^{(1)} \in \mathbb{Z}_\phi$ and $a^{(2)} \in \mathbb{Z}_\phi$, respectively, such that $a = a^{(1)} + a^{(2)} \mod \phi$. Users who fail this ZKP are excluded from subsequent computation.

2. Repeat the following steps until the ARPACK routine indicates convergence or stops after certain number of iterations:

   (a) **Consistency Check** When `dsaupd` returns control to $S_1$ with a vector, the server converts the vector to $v \in \mathbb{Z}_\phi^m$ and sends it to all users. The servers execute the consistency check protocol for each user.

   (b) **Aggregate** For any users who are marked as FAIL, or fail to respond, the servers simply ignore their data and exclude them from subsequent computation. $Q$ is updated accordingly. For this round they compute $s = \sum_{i \in Q} d_i$ and $S_1$ returns it as the matrix-vector product to `dsaupd` which runs another iteration.

3. **Output** $S_1$ outputs

$$\begin{aligned} \Sigma_k &= diag(\sigma_1, \sigma_2, \ldots, \sigma_k) \in \mathbb{R}^{k \times k} \\ V_k &= [v_1, v_2, \ldots, v_k,] \in \mathbb{R}^{m \times k} \end{aligned}$$

with $\sigma_i = \sqrt{\lambda_i}$ where $\lambda_i$ is the $i$th eigenvalue and $v_i$ the corresponding eigenvector computed by ARPACK, $i = 1, \ldots, k$, and $\lambda_1 \geq \lambda_2 \ldots \geq \lambda_k$.

For accuracy of the result produced by this protocol in the presence of *actively* cheating users, we have

**Theorem 3** *Let $n_c$ be the number of cheating users. We use $\tilde{\cdot}$ to denote perturbed quantity and $\sigma_i$ the $i$-th singular value of matrix $A$. Assuming that honest users vector L2-norms are uniformly random in $[0, L)$ and $n_c \ll n$, then*

$$\sqrt{\frac{\sum_i (\tilde{\sigma}_i - \sigma_i)^2}{\sum_i \sigma_i^2}} < 2\sqrt{\frac{n_c}{n}}$$

**Proof** The classic Weyl and Mirsky theorems [47] bound the perturbation to $A$'s singular values in terms of the Frobenius norm $\|\cdot\|_F$ of $E := A - \tilde{A}$:

$$\sqrt{\sum_i (\tilde{\sigma}_i - \sigma_i)^2} \leq \|E\|_F$$

In our case each row $a_i$ of $A$ is held by a user, we have

$$\|E\|_F = \sqrt{\sum_{i=1}^{n} \|\tilde{a}_i - a_i\|_2^2}$$

Since the protocol ensures that $\|a_i\|_2 < L$ for all users,

$$\sqrt{\sum_i (\tilde{\sigma}_i - \sigma_i)^2} \leq \sqrt{\sum_{i=1}^{n} \|\tilde{a}_i - a_i\|_2^2} < \sqrt{n_c} L$$

Let $\xi = \sqrt{\sum_i (\tilde{\sigma}_i - \sigma_i)^2} / \sqrt{\sum_i \sigma_i^2}$, and assuming that honest users vector L2-norms are uniformly random in $[0, L)$ and $n_c \ll n$, then

$$\xi = \frac{\sqrt{\sum_i (\tilde{\sigma}_i - \sigma_i)^2}}{\|A\|_F} < \frac{\sqrt{n_c} L}{0.5\sqrt{(n - n_c)} L} \approx 2\sqrt{\frac{n_c}{n}}$$

The scheme is also quite robust against users failures. During our tests reported in section 7, we simulated a fraction of random users "dropping out" of each iteration. Even when up to 50% of the users dropped, for all our test sets, the computation still converged without noticeable loss of accuracy, measured by residual error (see section 7.1) using the final matrix with failed users data ignored. This allows us to handle malicious users who actively try to disrupt the computation and those who fail to response due to technical problems (e.g., network failure) in a uniform way.

### 6.6 Privacy Analysis

Note that the protocol does not compute $U_k$. This is intentional. $U_k$ contains information about user data: the $i^{th}$ row of $U_k$ encodes user $i$'s data in the $k$-dimensional subspace and should not be revealed at all in a privacy-respecting application. $V_k$, on the other hand, encodes "item" data in the $k$-dimensional subspace (e.g., if $A$ is a user-by-movie rating matrix, the items will be movies). In most applications the desired information can be computed from the singular values ($\Sigma_k$) and the right singular vectors ($V_k^T$) (e.g., [11])

At each iteration, the protocol reveals the matrix-vector product $A^T A v$ for some vectors $v$. This is not a problem because the final results $\Sigma_k$ and $V_k^T$ already give an approximation of $A^T A$ ($A^T A = V \Sigma^2 V^T$). A simulator with the final results can approximate the intermediate sums. Therefore the intermediate aggregates do not reveal more information.

## 7 Implementation and Evaluation

The P4P framework, including the SVD protocol, has been implemented in Java using JNI and a NativeBigInteger implementation from I2P (http://www.i2p2.de/). We run several experiments. The server is a 2.50GHz Xeon E5420 with 32GB memory, the clients are 2.00GHz Xeon E5405 with 800 MB memory allocated to the tests. In all the experiments, $\phi$ is set to be a 62-bit integer and $q$ 1024-bit.

We evaluated our implementation on three data sets: the Enron Email Data set [14], EachMovie (EM), and a randomly generated dense matrix (RAND). The Enron corpus contains email data from 150 users, spanning a period of about 5 years (Jan. 1998 to Dec 2002). Our test was run on the social graph defined by the email communications. The graph is represented as a $150 \times 150$ matrix $A$ with $A(i, j)$ being the number of emails sent by user $i$ to user $j$. EachMovie is a well-known test data set for collaborative filtering. It comprises ratings of 1648 movies by 74424 users. Each rating is a number in the range $[0, 1]$. Both the Enron and EachMovie data sets are very sparse, with densities 0.0736 and 0.0229, respectively. To test the performance of our protocol on dense matrices, we generated randomly a $2000 \times 2000$ matrix with entries chosen in the range $[-2^{20}, 2^{20}]$.

### 7.1 Precision and Round Complexity

We measured two quantities: $N$, the number of IRAM iterations until ARPACK indicates convergence, and $\epsilon$, the relative error. $N$ is the number of matrix-vector computation that was required for the ARPACK to converge. It is also the number of times P4P aggregation is invoked. The error $\epsilon$ measures the maximum relative residual norm among all eigenpairs computed:

$$\epsilon = \max_{i=1,\ldots,k} \frac{\|A^T A v_i - \lambda_i v_i\|_2}{\|v_i\|_2}$$

Table 2 summarizes the results. In all these tests, we used machine precision as the tolerance input to ARPACK. The accuracy we obtained is very good: $\epsilon$ remains very small for all tests ($10^{-12}$ to $10^{-8}$). In terms of round complexity, $N$ ranges from under 100 to a few hundreds. For comparison, we also measured the number of iterations required by ARPACK when we perform the matrix-vector multiplication directly without the P4P aggregation. In all experiments, we found no difference in $N$ between this direct method and our private implementation.

### 7.2 Performance

We measured both running time and communication cost of our scheme. We focused on server load since each user only needs to handle her own data so is not a bottleneck. We first present the case with $\kappa = 2$ servers. We measured the work on the server hosting the ARPACK engine since it shares more load.

First, the implementation confirmed our observations about the difference in costs for manipulating large and small integers. With 1024-bit key length, one exponentiation within the multiplicative group $\mathbb{Z}_q^*$ takes 5.86 milliseconds. Addition and multiplication of two numbers, also within the group, take 0.024 and 0.062 milliseconds, respectively. In contrast, adding two 64-bit integers, which is the basic operations P4P framework performs, needs only $2.7 \times 10^{-6}$ milliseconds. The product ZKP takes 35.7 ms verifier time and 24.3 ms prover time. The equivalence ZKP takes no time since it is simply revealing the difference of the two random numbers used in the commitments [45]. For each consistency check, the user needs to compute 9 commitments, 3 product ZKPs, 1 equivalence ZKP and 4 large integer multiplications. The total cost is 178.63 milliseconds for each user. For every user, each server needs to spend 212.83 milliseconds on verification.

For our test data sets, it takes 74.73 seconds of server time to validate and aggregate all 150 Enron users data on a *single* machine (each user needs to spend 726 milliseconds to prepare the zero-knowledge proofs). This translates into a total of 5000 seconds or 83 minutes spent on private P4P aggregation to compute $k = 10$ singular-pairs. To compute the same number of singular pairs for EachMovie, aggregating all users data takes about 6 hours (again on a single machine) and the total time for 70 rounds is 420 hours. Note that the total includes *both* verification and computation so it is the cost of a complete run. The server load appears large but actually is very inexpensive. The aggregation process is trivially parallelizable and using a cluster of, say 200 nodes, will reduce the running time to about 2 hours. This amounts to a very insignificant cost for most service providers: Using Amazon EC2's price as a benchmark, it costs $0.80 per hour for 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each). Data transfer price is $0.100 per GB. The total cost for comput-

Table 2: Round Complexity and Precision

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Enron | $k$ | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| | $N$ | 67 | 97 | 122 | 162 | 109 | 137 | 172 | 167 | 171 | 169 |
| | $\epsilon(\times 10^{-8})$ | 0.00049 | 0.0021 | 0.0046 | 0.0084 | 0.0158 | 0.0452 | 0.121 | 0.266 | 0.520 | 1.232 |
| EM | $k$ | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| | $N$ | 70 | 140 | 254 | 222 | 276 | 371 | 322 | 356 | 434 | 508 |
| | $\epsilon(\times 10^{-12})$ | 0.470 | 0.902 | 1.160 | 1.272 | 1.526 | 1.649 | 1.687 | 2.027 | 2.124 | 2.254 |
| RAND | $k$ | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| | $N$ | 304 | 404 | 450 | 480 | 550 | 700 | 770 | 720 | 810 | 800 |
| | $\epsilon(\times 10^{-9})$ | 3.996 | 3.996 | 3.996 | 3.996 | 3.996 | 3.996 | 3.996 | 3.996 | 3.996 | 3.996 |

ing SVD for a system with 74424 users is merely about $15, including data transfer and adjusted for difference in CPU performance between our experiments and EC2.

To compare with alternative solutions, we implemented a method based on homomorphic encryption which is a popular private data mining technique (see e.g., [11, 51]). We did not try other methods, such as the "add/subtract random" approach, with players adding their values to a running total, because they do not allow for verification of user data thus are insecure in our model. We tested both ElGamal and Paillier encryptions with the same security parameter as our P4P experiments (i.e., 1024-bit key). With the homomorphic encryption approach, it is almost impossible to execute the ZK verification (although there is a protocol [11]) as it takes hours to verify one user. So we only compared the time needed for computing the aggregates. Figure 2 shows the ratios of running time between homomorphic encryption and P4P for SVD on the three data sets. P4P is at least 8 orders of magnitude faster in all cases for both ElGamal and Paillier. And this translates to tens of millions of dollars of cost for the homomorphic encryption schemes if the computation is done using Amazon's EC2 service not even counting data transfer expenses.

The communication overhead is also very small since the protocol passes very few large integers. The extra communication per client for one L2-norm ZKP is under 50 kilobytes, and under 100 bytes for the consistency check, while other solutions require some hundreds of megabytes. This is significantly smaller than the size of an average web page. The additional workload for the server is less than serving an extra page to each user.

**The case with $\kappa > 2$ servers**: Although we do not expect the scheme to be deployed with a large number of servers, we provide some analysis here in case stronger protection is required. Each server's work can be divided into two parts: processing clients and communicating with other servers. Most expensive interactions are with the clients (including verifying the ZKPs etc.), which can be performed on a single server and is independent of $\kappa$. The interaction among servers is simply data exchange and there is no complex computation involved.

Data exchange among the servers serves two purposes:

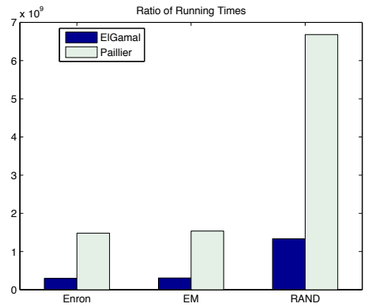



Figure 2: Running time ratios between homomorphic encryption based solutions and P4P.

reconstructing shared secrets when necessary (the final sum in the end of each iteration and the commitments during the verification) and reaching agreement regarding a user's status (each server needs to verify that the user computes a share of the commitments correctly). And since each server is semi-honest, for the second part they only need pass the final conclusion, verification of the ZKPs can be done on only one of the servers.

For constructing the final sum, all servers must send their shares to the server hosting ARPACK. The later will receive a total of $8\kappa m$ bytes (assuming data is encoded using double precision) which is about $8\kappa$ MB if $m = 10^6$. For the consistency check, during each iteration, one server is selected as the "master". All other servers sends their shares of the commitments to the master. This includes $3n$ large integers in $\mathbb{Z}_q$ (3 for each user) from each server. In addition, each non-master server also sends to the master an $n$-bit bitmap, encoding whether each user computes the commitments to the shares correctly. The master will reconstruct the complete commitments and verify the ZKPs. It then broadcasts an $n$-bit bitmap encoding whether each user passes the consistency check to all other servers. For the master, the total communication cost is receiving $3n(\kappa-1)$ integers in $\mathbb{Z}_q$ and $\kappa n$-bit strings and sending $(\kappa-1)n$ bits. With $n = 10^6$ and $|q| = 1024$, these amount to 384 $(\kappa-1)$ MB and approximately 0.1 $(\kappa-1)$ MB, respectively. For other servers, the sending and receiving costs are approximately 384 MB and 0.1 MB, respectively. We believe such cost is practical for small $\kappa$ (e.g., 3 or 4). Note that the master does not have to be collocated with the ARPACK engine so the servers can take turns to serve as the master to share the load.

As for the computation associated with using $\kappa$ servers (the part that is independent of $\kappa$ has been discussed earlier and omitted here), the master needs to perform $3n(\kappa-1)$ multiplications in $\mathbb{Z}_q^*$. Using our benchmark, this amounts to $0.186(\kappa-1)$ seconds for $n = 10^6$ users. Again we believe this is practical for small $\kappa$. The other servers do not need to do any extra work.

### 7.3 Scalability

We also experimented with a few very large matrices, with dimensionality ranging from tens of thousands to over a hundred million. They are document-term or user-query matrices that are used for latent semantic analysis. To facilitate the tests, we did not include the data verification ZKPs, as our previous benchmarks show they amount to an insignificant fraction of the cost. Due to space and resource limit we did not test how performance varies with dimensionality and other parameters. Rather, these results are meant to demonstrate the capability of our system, which we have shown to maintain privacy at very low cost, to handle large data sets at various configurations.

Table 3 summarizes some of the results. The running time measures the time of a complete run, i.e., from the start of the job till the results are safely written to disk. It includes both the computation time of the server (including the time spent on invoking the ARPACK engine) and the clients (which are running in parallel), and the communication time. In the table, frontend processors refer to the machines that interact with the users directly. Large-scale systems usually use multiple frontend machines, each serving a subset of the users. This is also a straightforward way to parallelize the aggregation process, i.e., each frontend machine receives data from a subset of users and aggregates them before forwarding to the server. On one hand, the more frontend machines the faster the sub-aggregates can be computed. On the other hand, the server's communication cost is linear in the number of frontend processors. The optimal solution must strike a balance between the two. Due to resource limitation, we were not able to use the optimal configuration for all our tests. The results are feasible even in these sub-optimal cases.

## 8 Conclusion

In this paper we present a new framework for privacy-preserving distributed data mining. Our protocol is based on secret sharing over *small* field, achieving orders of magnitude reduction in running time over alternative solutions with large-scale data. The framework also admits very efficient zero-knowledge tools that can be used to verify user data. They provide practical solutions for handling cheating users. P4P demonstrates that cryptographic building blocks can work harmoniously with existing tools, providing privacy without degrading their efficiency. Most components described in this paper have been implemented and the source code is available at http://bid.berkeley.edu/projects/p4p/. Our goal is to make it a useful tool for developers in data mining and others to build privacy preserving real-world applications.

## References


[1] ALDERMAN, E., AND KENNEDY, C. *The Right to Privacy*. DIANE Publishing Co., 1995.

[2] BEAVER, D., AND GOLDWASSER, S. Multiparty computation with faulty majority. In *CRYPTO '89*.

[3] BEERLIOVÁ-TRUBÍNIOVÁ, Z., AND HIRT, M. Perfectly-secure mpc with linear communication complexity. In *TCC 2008* (2008), Springer-Verlag, pp. 213–230.

[4] BEIMEL, A., NISSIM1, K., AND OMRI, E. Distributed private data analysis: Simultaneously solving how and what. In *CRYPTO 2008*.

[5] BEN-DAVID, A., NISAN, N., AND PINKAS, B. Fairplaymp: a system for secure multi-party computation. In *CCS '08* (2008), ACM, pp. 257–266.

[6] BEN-OR, M., GOLDWASSER, S., AND WIGDERSON, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC'88* (1988), ACM, pp. 1–10.

[7] BLUM, A., DWORK, C., MCSHERRY, F., AND NISSIM, K. Practical privacy: the SuLQ framework. In *PODS '05* (2005), ACM Press, pp. 128–138.

[8] BLUM, A., LIGETT, K., AND ROTH, A. A learning theory approach to non-interactive database privacy. In *STOC 08*.

[9] BOAZ BARAK, E. A. Privacy, accuracy, and consistency too: a holistic solution to contingency table release. In *PODS '07*.

[10] CANNY, J. Collaborative filtering with privacy via factor analysis. In *SIGIR '02*.

[11] CANNY, J. Collaborative filtering with privacy. In *IEEE Symposium on Security and Privacy* (2002), pp. 45–57.

[12] CHEN, H., AND CRAMER, R. Algebraic geometric secret sharing schemes and secure multi-party computations over small fields. In *CRYPTO 2006*.

[13] CHU, C.-T., KIM, S. K., LIN, Y.-A., YU, Y., BRADSKI, G., NG, A. Y., AND OLUKOTUN, K. Map-reduce for machine learning on multicore. In *NIPS 2006* (2006).

[14] COHEN, W. W. Enron email dataset. http://www-2.cs.cmu.edu/~enron/.

[15] CRAMER, R., AND DAMGÅRD, I. Zero-knowledge proof for finite field arithmetic, or: Can zero-knowledge be for free? In *CRYPTO '98* (1998), Springer-Verlag.

Table 3: SVD of Large Matrices

| $n$ | $m$ | $k$ | No. Frontend Processors | Time (hours) | Iterations |
|---|---|---|---|---|---|
| 100,443 | 176,573 | 200 | 32 | 1.4 | 1287 |
| 12,046,488 | 440,208 | 200 | 128 | 6.0 | 354 |
| 149,519,201 | 478,967 | 250 | 128 | 8.3 | 1579 |
| 37,389,030 | 366,881 | 300 | 128 | 9.1 | 1839 |
| 1,363,716 | 2,611,186 | 200 | 1 | 14.8 | 1260 |
| 33,193,487 | 1,949,789 | 200 | 128 | 28.0 | 1470 |


[16] DAMGÅRD, I., ISHAI, Y., KRØIGAARD, M., NIELSEN, J. B., AND SMITH, A. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO 2008* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 241–261.

[17] DAS, A. S., DATAR, M., GARG, A., AND RAJARAM, S. Google news personalization: scalable online collaborative filtering. In *WWW '07* (2007), ACM Press, pp. 271–280.

[18] DHANJANI, N. Amazon's elastic compute cloud [ec2]: Initial thoughts on security implications. http://www.dhanjani.com/archives/2008/04/.

[19] DINUR, I., AND NISSIM, K. Revealing information while preserving privacy. In *PODS '03* (2003), pp. 202–210.

[20] DUAN, Y. Privacy without noise. In *CIKM '09*.

[21] DUAN, Y., AND CANNY, J. Practical private computation and zero-knowledge tools for privacy-preserving distributed data mining. In *SDM '08* (2008).

[22] DUAN, Y., WANG, J., KAM, M., AND CANNY, J. A secure online algorithm for link analysis on weighted graph. In *Proc. of the Workshop on Link Analysis, Counterterrorism and Security, SDM 05*, pp. 71–81.

[23] DWORK, C. Ask a better question, get a better answer a new approach to private data analysis. In *ICDT 2007* (2007), Springer, pp. 18–27.

[24] DWORK, C., KENTHAPADI, K., MCSHERRY, F., MIRONOV, I., AND NAOR, M. Our data, ourselves: Privacy via distributed noise generation. In *EUROCRYPT 2006* (2006), Springer.

[25] DWORK, C., MCSHERRY, F., NISSIM, K., AND SMITH, A. Calibrating noise to sensitivity in private data analysis. In *TCC 2006* (2006), Springer, pp. 265–284.

[26] FEIGENBAUM, J., NISAN, N., RAMACHANDRAN, V., SAMI, R., AND SHENKER, S. Agents' privacy in distributed algorithmic mechanisms. In *Workshop on Economics and Information Securit* (Berkeley, CA, May 2002).

[27] FIAT, A., AND SHAMIR, A. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO 86*.

[28] FITZI, M., HIRT, M., AND MAURER, U. General adversaries in unconditional multi-party computation. In *ASIACRYPT ' 99*.

[29] GENNARO, R., RABIN, M. O., AND RABIN, T. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *PODC '98*, pp. 101–111.

[30] GOLDREICH, O. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.

[31] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game. In *STOC '87*.

[32] GOLDREICH, O., AND OREN, Y. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology 7*, 1 (1994), 1–32.

[33] GOLDWASSER, S., AND LEVIN, L. Fair computation of general functions in presence of immoral majority. In *CRYPTO '90* (1991), Springer-Verlag, pp. 77–93.

[34] HIRT, M., AND MAURER, U. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In *PODC '97*.

[35] HIRT, M., AND MAURER, U. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology 13*, 1 (2000), 31–60.

[36] KEARNS, M. Efficient noise-tolerant learning from statistical queries. In *STOC '93* (1993), pp. 392–401.

[37] LEHOUCQ, R. B., SORENSEN, D. C., AND YANG, C. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, 1998.

[38] LINDELL, Y., AND PINKAS, B. Privacy preserving data mining. *Journal of cryptology 15*, 3 (2002), 177–206.

[39] LINDELL, Y., PINKAS, B., AND SMART, N. P. Implementing two-party computation efficiently with security against malicious adversaries. In *SCN '08*.

[40] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay—a secure two-party computation system. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 20–20.

[41] MCSHERRY, F., AND MIRONOV, I. Differentially private recommender systems: Building privacy into the netflix prize contenders. In *KDD '09*.

[42] MCSHERRY, F., AND TALWAR, K. Mechanism design via differential privacy. In *FOCS '07*.

[43] NISSIM, K., RASKHODNIKOVA, S., AND SMITH, A. Smooth sensitivity and sampling in private data analysis. In *STOC '07* (2007), ACM, pp. 75–84.

[44] PAILLIER, P. Trapdooring discrete logarithms on elliptic curves over rings. In *ASIACRYPT '00*.

[45] PEDERSEN, T. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO '91*.

[46] PINKAS, B., SCHNEIDER, T., SMART, N., AND WILLIAMS, S. Secure two-party computation is practical. Cryptology ePrint Archive, Report 2009/314, 2009.

[47] STEWART, G. W., AND SUN, J.-G. *Matrix Perturbation Theory*. Academic Press, 1990.

[48] TREFETHEN, L. N., AND III, D. B. *Numerical Linear Algebra*. SIAM, 1997.

[49] VAIDYA, J., AND CLIFTON, C. Privacy-preserving k-means clustering over vertically partitioned data. In *KDD '03*.

[50] WRIGHT, R., AND YANG, Z. Privacy-preserving bayesian network structure computation on distributed heterogeneous data. In *KDD '04* (2004), pp. 713–718.

[51] YANG, Z., ZHONG, S., AND WRIGHT, R. N. Privacy-preserving classification of customer data without loss of accuracy. In *SDM 2005* (2005).

[52] YAO, A. C.-C. Protocols for secure computations. In *FOCS '82* (1982), IEEE, pp. 160–164.


## Notes

[1] Most mining algorithms need to bound the amount of noise in the data to produce meaningful results. This means that the fraction of cheating users is usually below a much lower threshold (e.g. $\alpha < 20\%$).

# SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics

Martin Burkhart, Mario Strasser, Dilip Many, Xenofontas Dimitropoulos
*ETH Zurich, Switzerland*
{burkhart, strasser, dmany, fontas}@tik.ee.ethz.ch

## Abstract

Secure multiparty computation (MPC) allows joint privacy-preserving computations on data of multiple parties. Although MPC has been studied substantially, building solutions that are practical in terms of computation and communication cost is still a major challenge. In this paper, we investigate the practical usefulness of MPC for multi-domain network security and monitoring. We first optimize MPC comparison operations for processing high volume data in near real-time. We then design privacy-preserving protocols for event correlation and aggregation of network traffic statistics, such as addition of volume metrics, computation of feature entropy, and distinct item count. Optimizing performance of parallel invocations, we implement our protocols along with a complete set of basic operations in a library called SEPIA. We evaluate the running time and bandwidth requirements of our protocols in realistic settings on a local cluster as well as on PlanetLab and show that they work in near real-time for up to 140 input providers and 9 computation nodes. Compared to implementations using existing general-purpose MPC frameworks, our protocols are significantly faster, requiring, for example, 3 minutes for a task that takes 2 days with general-purpose frameworks. This improvement paves the way for new applications of MPC in the area of networking. Finally, we run SEPIA's protocols on real traffic traces of 17 networks and show how they provide new possibilities for distributed troubleshooting and early anomaly detection.

## 1 Introduction

A number of network security and monitoring problems can substantially benefit if a group of involved organizations aggregates private data to jointly perform a computation. For example, IDS alert correlation, e.g., with DOMINO [49], requires the joint analysis of private alerts. Similary, aggregation of private data is useful for alert signature extraction [30], collaborative anomaly detection [34], multi-domain traffic engineering [27], detecting traffic discrimination [45], and collecting network performance statistics [42]. All these approaches use either a trusted third party, e.g., a university research group, or peer-to-peer techniques for data aggregation and face a delicate privacy versus utility tradeoff [32]. Some private data typically have to be revealed, which impedes privacy and prohibits the acquisition of many data providers, while data anonymization, used to remove sensitive information, complicates or even prohibits developing good solutions. Moreover, the ability of anonymization techniques to effectively protect privacy is questioned by recent studies [29]. One possible solution to this privacy-utility tradeoff is MPC.

For almost thirty years, MPC [48] techniques have been studied for solving the problem of jointly running computations on data distributed among multiple organizations, while provably preserving data privacy without relying on a trusted third party. In theory, any computable function on a distributed dataset is also securely computable using MPC techniques [20]. However, designing solutions that are practical in terms of running time and communication overhead is non-trivial. For this reason, MPC techniques have mainly attracted theoretical interest in the last decades. Recently, optimized basic primitives, such as comparisons [14, 28], make progressively possible the use of MPC in real-world applications, e.g., an actual sugar-beet auction [7] was demonstrated in 2009.

Adopting MPC techniques to network monitoring and security problems introduces the important challenge of dealing with voluminous input data that require online processing. For example, anomaly detection techniques typically require the online generation of traffic volume and distributions over port numbers or IP address ranges. Such input data impose stricter requirements on the performance of MPC protocols than, for example, the input bids of a distributed MPC auction [7]. In particular, network monitoring protocols should process potentially

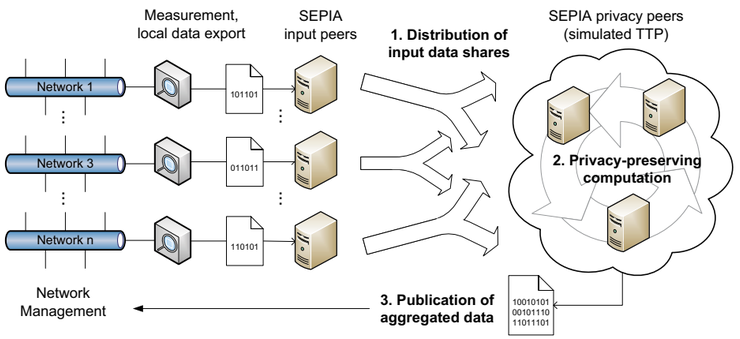



Figure 1: Deployment scenario for SEPIA.

thousands of input values while meeting *near real-time* guarantees[1]. This is not presently possible with existing general-purpose MPC frameworks.

In this work, we design, implement, and evaluate SEPIA (Security through Private Information Aggregation), a library for efficiently aggregating multi-domain network data using MPC. The foundation of SEPIA is a set of optimized MPC operations, implemented with performance of parallel execution in mind. By not enforcing protocols to run in a constant number of rounds, we are able to design MPC comparison operations that require up to 80 times less distributed multiplications and, amortized over many parallel invocations, run much faster than constant-round alternatives. On top of these comparison operations, we design and implement novel MPC protocols tailored for network security and monitoring applications. The *event correlation* protocol identifies events, such as IDS or firewall alerts, that occur frequently in multiple domains. The protocol is generic having several applications, for example, in alert correlation for early exploit detection or in identification of multi-domain network traffic heavy-hitters. In addition, we introduce SEPIA's *entropy* and *distinct count* protocols that compute the entropy of traffic feature distributions and find the count of distinct feature values, respectively. These metrics are used frequently in traffic analysis applications. In particular, the entropy of feature distributions is used commonly in anomaly detection, whereas distinct count metrics are important for identifying scanning attacks, in firewalls, and for anomaly detection. We implement these protocols along with a vector addition protocol to support additive operations on time-series and histograms.

A typical setup for SEPIA is depicted in Fig. 1 where individual networks are represented by one *input peer* each. The input peers distribute shares of secret input data among a (usually smaller) set of *privacy peer*s using Shamir's secret sharing scheme [40]. The privacy peers perform the actual computation and can be hosted by a subset of the networks running input peers but also by external parties. Finally, the aggregate computation result is sent back to the networks. We adopt the semi-honest adversary model, hence privacy of local input data is guaranteed as long as the majority of privacy peers is honest. A detailed description of our security assumptions and a discussion of their implications is presented in Section 4.

Our evaluation of SEPIA's performance shows that SEPIA runs in near real-time even with 140 input and 9 privacy peers. Moreover, we run SEPIA on traffic data of 17 networks collected during the global Skype outage in August 2007 and show how the networks can use SEPIA to troubleshoot and timely detect such anomalies. Finally, we discuss novel applications in network security and monitoring that SEPIA enables. In summary, this paper makes the following contributions:

1. We introduce efficient MPC comparison operations, which outperform constant-round alternatives for many parallel invocations.
2. We design novel MPC protocols for event correlation, entropy and distinct count computation.
3. We introduce the SEPIA library, in which we implement our protocols along with a complete set of basic operations, optimized for parallel execution. SEPIA is made publicly available [39].
4. We extensively evaluate the performance of SEPIA on realistic settings using synthetic and real traces and show that it meets near real-time guarantees even with 140 input and 9 privacy peers.
5. We run SEPIA on traffic from 17 networks and show how it can be used to troubleshoot and timely detect anomalies, exemplified by the Skype outage.

The paper is organized as follows: We specify the computation scheme in the next section and present our optimized comparison operations in Section 3. In Section 4, we specify our adversary model and security assumptions, and build the protocols for event correlation, vector addition, entropy, and distinct count computation. We evaluate the protocols and discuss SEPIA's design in Sections 5 and 6, respectively. Then, in Section 7 we outline SEPIA's applications and conduct a case study on real network data that demonstrates SEPIA's benefits in distributed troubleshooting and early anomaly detection. Finally, we discuss related work in Section 8 and conclude our paper in Section 9.

## 2 Preliminaries

Our implementation is based on Shamir secret sharing [40]. In order to *share* a secret value $s$ among a set of $m$ players, the dealer generates a random polynomial $f$ of degree $t = \lfloor (m-1)/2 \rfloor$ over a prime field $\mathbb{Z}_p$ with $p > s$, such that $f(0) = s$. Each player $i = 1 \dots m$ then receives an evaluation point $s_i = f(i)$ of $f$. $s_i$ is called the share of player $i$. The secret $s$ can be reconstructed from any $t+1$ shares using Lagrange interpolation but is completely undefined for $t$ or less shares. To actually *reconstruct* a secret, each player sends his shares to all other players. Each player then locally interpolates the secret. For simplicity of presentation, we use $[s]$ to denote the vector of shares $(s_1, \dots, s_m)$ and call it a *sharing* of $s$. In addition, we use $[s]_i$ to refer to $s_i$. Unless stated otherwise, we choose $p$ with 62 bits such that arithmetic operations on secrets and shares can be performed by CPU instructions directly, not requiring software algorithms to handle big integers.

**Addition and Multiplication** Given two sharings $[a]$ and $[b]$, we can perform private addition and multiplication of the two values $a$ and $b$. Because Shamir's scheme is linear, addition of two sharings, denoted by $[a] + [b]$, can be computed by having each player locally add his shares of the two values: $[a+b]_i = [a]_i + [b]_i$. Similarly, local shares are subtracted to get a share of the difference. To add a public constant $c$ to a sharing $[a]$, denoted by $[a] + c$, each player just adds $c$ to his share, i.e., $[a+c]_i = [a]_i + c$. Similarly, for multiplying $[a]$ by a public constant $c$, denoted by $c[a]$, each player multiplies its share by $c$. Multiplication of two sharings requires an extra round of communication to guarantee randomness and to correct the degree of the new polynomial [4, 19]. In particular, to compute $[a][b] = [ab]$, each player first computes $d_i = [a]_i[b]_i$ locally. He then shares $d_i$ to get $[d_i]$. Together, the players then perform a distributed Lagrange interpolation to compute $[ab] = \sum_i \lambda_i [d_i]$ where $\lambda_i$ are the Lagrange coefficients. Thus, a distributed multiplication requires a synchronization round with $m^2$ messages, as each player $i$ sends to each player $j$ the share $[d_i]_j$. To specify protocols, composed of basic operations, we use a shorthand notation. For instance, we write $foo([a], b) := ([a] + b)([a] + b)$, where $foo$ is the protocol name, followed by input parameters. Valid input parameters are sharings and public constants. On the right side, the function to be computed is given, a binomial in that case. The output of $foo$ is again a sharing and can be used in subsequent computations. All operations in $\mathbb{Z}_p$ are performed modulo $p$, therefore $p$ must be large enough to avoid modular reductions of intermediate results, e.g., if we compute $[ab] = [a][b]$, then $a$, $b$, and $ab$ must be smaller than $p$.

**Communication** A set of independent multiplications, e.g., $[ab]$ and $[cd]$, can be performed in parallel in a single round. That is, intermediate results of all multiplications are exchanged in a single synchronization step. *A round simply is a synchronization point where players have to exchange intermediate results* in order to continue computation. While the specification of the protocols is synchronous, we do not assume the network to be synchronous during runtime. In particular, the Internet is better modeled as asynchronous, not guaranteeing the delivery of a message before a certain time. Because we assume the semi-honest model, we only have to protect against high delays of individual messages, potentially leading to a reordering of message arrival. In practice, we implement communication channels using SSL sockets over TCP/IP. TCP applies acknowledgments, timeouts, and sequence numbers to preserve message ordering and to retransmit lost messages, providing FIFO channel semantics. We implement message synchronization in parallel threads to minimize waiting time. Each player proceeds to the next round immediately after sending and receiving all intermediate values.

**Security Properties** All the protocols we devise are compositions of the above introduced addition and multiplication primitives, which were proven correct and *information-theoretically* secure by Ben-Or, Goldwasser, and Wigderson [4]. In particular, they showed that in the semi-honest model, where adversarial players follow the protocol but try to learn as much as possible by sharing the information they received, no set of $t$ or less corrupt players gets any additional information other than the final function value. Also, these primitives are *universally composable*, that is, the security properties remain intact under stand-alone and concurrent composition [11]. Because the scheme is information-theoretically secure, i.e., it is secure against computationally unbounded adversaries, the confidentiality of secrets does not depend on the field size $p$. For instance, regarding confidentiality, sharing a secret $s$ in a field of size $p > s$ is equivalent to sharing each individual bit of $s$ in a field of size $p = 2$. Because we use SSL for implementing secure channels, the *overall system* relies on PKI and is only computationally secure.

## 3 Optimized Comparison Operations

Unlike addition and multiplication, comparison of two shared secrets is a very expensive operation. Therefore, we now devise optimized protocols for equality check, less-than comparison and a short range check. The complexity of an MPC protocol is typically assessed counting the number of distributed multiplications and rounds, because addition and multiplication with public values only require local computation. Damgård *et al.* introduced the bit-decomposition protocol [14] that achieves comparison by decomposing shared secrets into a shared bit-wise representation. On shares of individual bits, comparison is straight-forward. With $l = \log_2(p)$, the protocols in [14] achieve a comparison with $205l + 188l \log_2 l$ multiplications in 44 rounds and equality test with $98l + 94l \log_2 l$ multiplications in 39 rounds. Subsequently, Nishide and Ohta [28] have improved these protocols by not decomposing the secrets but using bitwise shared random numbers. They do comparison with $279l + 5$ multiplications in 15 rounds and equality test with $81l$ multiplications in 8 rounds. While these are constant-round protocols as preferred in theoretical research, they still involve lots of multiplications. For instance, an equality check of two shared IPv4 addresses ($l = 32$) with the protocols in [28] requires 2592 distributed multiplications, each triggering $m^2$ messages to be transmitted over the network.

**Constant-round vs. number of multiplications** Our key observation for improving efficiency is the following: For scenarios with many parallel protocol invocations it is possible to build much more practical protocols by not enforcing the constant-round property. Constant-round means that the number of rounds does not depend on the input parameters. We design protocols that run in O($l$) rounds and are therefore not constant-round, although, once the field size $p$ is defined, the number of rounds is also fixed, i.e., not varying at runtime. The overall local running time of a protocol is determined by i) the local CPU time spent on computations, ii) the time to transfer intermediate values over the network, and iii) delay experienced during synchronization. Designing constant-round protocols aims at reducing the impact of iii) by keeping the number of rounds fixed and usually small. To achieve this, high multiplicative constants for the number of multiplications are often accepted (e.g., $279l$). Yet, both i) and ii) directly depend on the number of multiplications. For applications with few parallel operations, protocols with few rounds (usually constant-round) are certainly faster. However, with many parallel operations, as required by our scenarios, the impact of network delay is amortized and the number of multiplications (the actual workload) becomes the dominating factor. Our evaluation results in Section 5.1 and 5.4 confirm this and show that CPU time and network bandwidth are the main constraining factors, calling for a reduction of multiplications.

**Equality Test** In the field $\mathbb{Z}_p$ with $p$ prime, Fermat's little theorem states

$$c^{p-1} = \begin{cases} 0 & \text{if } c = 0 \\ 1 & \text{if } c \neq 0 \end{cases} \quad (1)$$

Using (1) we define a protocol for equality test as follows:

$$equal([a], [b]) := 1 - ([a] - [b])^{p-1}$$

The output of $equal$ is [1] in case of equality and [0] otherwise and can hence be used in subsequent computations. Using square-and-multiply for the exponentiation, we implement $equal$ with $l + k - 2$ multiplications in $l$ rounds, where $k$ denotes the number of bits set to 1 in $p - 1$. By using carefully picked prime numbers with $k \leq 3$, we reduce the number of multiplications to $l + 1$. In the above example for comparing IPv4 addresses, this reduces the multiplication count by a factor of 76 from 2592 to 34.

Besides having few 1-bits, $p$ must be bigger than the range of shared secrets, i.e., if 32-bit integers are shared, an appropriate $p$ will have at least 33 bits. For any secret size below 64 bits it is easy to find appropriate $p$s with $k \leq 3$ within 3 additional bits.

**Less Than** For less-than comparison, we base our implementation on Nishide's protocol [28]. However, we apply modifications to again reduce the overall number of required multiplications by more than a factor of 10. Nishide's protocol is quite comprehensive and built on a stack of subprotocols for least-significant bit extraction (LSB), operations on bitwise-shared secrets, and (bitwise) random number sharing. The protocol uses the observation that $a < b$ is determined by the three predicates $a < p/2$, $b < p/2$, and $a - b < p/2$. Each predicate is computed by a call of the LSB protocol for $2a$, $2b$, and $2(a - b)$. If $a < p/2$, no wrap-around modulo $p$ occurs when computing $2a$, hence $LSB(2a) = 0$. However, if $a > p/2$, a wrap-around will occur and $LSB(2a) = 1$. Knowing one of the predicates in advance, e.g., because $b$ is not secret but publicly known, saves one of the three LSB calls and hence $1/3$ of the multiplications.

Due to space restrictions we omit to reproduce the entire protocol but focus on the modifications we apply. An important subprotocol in Nishide's construction is $PrefixOr$. Given a sequence of shared bits $[a_1], \ldots, [a_l]$ with $a_i \in \{0, 1\}$, $PrefixOr$ computes the sequence $[b_1], \ldots, [b_l]$ such that $b_i = \vee_{j=1}^{i} a_j$. Nishide's $PrefixOr$ requires only 7 rounds but $17l$ multiplications. We implement $PrefixOr$ based on the fact that $b_i = b_{i-1} \vee a_i$ and $b_1 = a_1$. The logical OR ($\vee$) can be computed using a single multiplication: $[x] \vee [y] = [x] + [y] - [x][y]$. Thus, our $PrefixOr$ requires $l - 1$ rounds and only $l - 1$ multiplications.

Without compromising security properties, we replace the $PrefixOr$ in Nishide's protocol by our optimized version and call the resulting comparison protocol $lessThan$. A call of $lessThan([a], [b])$ outputs [1] if $a < b$ and [0] otherwise. The overall complexity of $lessThan$ is $24l + 5$ multiplications in $2l + 10$ rounds as compared to Nishide's version with $279l + 5$ multiplications in 15 rounds.

**Short Range Check** To further reduce multiplications for comparing small numbers, we devise a check for short ranges, based on our $equal$ operation. Consider one wanted to compute $[a] < T$, where T is a small public constant, e.g., $T = 10$. Instead of invoking $lessThan([a], T)$ one can simply compute the polynomial $[\phi] = [a]([a] - 1)([a] - 2) \ldots ([a] - (T - 1))$. If the value of $a$ is between 0 and $T - 1$, exactly one term of $[\phi]$ will be zero and hence $[\phi]$ will evaluate to [0]. Otherwise, $[\phi]$ will be non-zero. Based on this, we define a protocol for checking short public ranges that returns [1] if $x \leq [a] \leq y$ and [0] otherwise:

$$shortRange([a], x, y) := equal\big(0, \prod_{i=x}^{y} ([a] - i)\big)$$

The complexity of $shortRange$ is $(y - x) + l + k - 2$ multiplications in $l + \log_2(y - x)$ rounds. Computing $lessThan([a], y)$ requires $16l + 5$ multiplications (1/3 is saved because $y$ is public). Hence, regarding the number of multiplications, computing $shortRange([a], 0, y - 1)$ instead of $lessThan([a], y)$ is beneficial roughly as long as $y \leq 15l$.

## 4 SEPIA Protocols

In this section, we compose the basic operations defined above into full-blown protocols for network event correlation and statistics aggregation. Each protocol is designed to run on continuous streams of input traffic data partitioned into time windows of a few minutes. For sake of simplicity, the protocols are specified for a single time window. We first define the basic setting of SEPIA protocols as illustrated in Fig. 1 and then introduce the protocols successively.

Our system has a set of $n$ users called *input peers*. The input peers want to jointly compute the value of a public function $f(x_1, \ldots, x_n)$ on their private data $x_i$ without disclosing anything about $x_i$. In addition, we have $m$ players called *privacy peers* that perform the computation of $f()$ by simulating a trusted third party (TTP). Each entity can take both roles, acting only as an input peer, privacy peer (PP) or both.

**Adversary Model and Security Assumptions** We use the semi-honest (a.k.a. honest-but-curious) adversary model for privacy peers. That is, honest privacy peers follow the protocol and do not combine their information. Semi-honest privacy peers do follow the protocol but try to infer as much as possible from the values (shares) they learn, also by combining their information. The privacy and correctness guarantees provided by our protocols are determined by Shamir's secret sharing scheme. In particular, the protocols are secure for $t < m/2$ semi-honest privacy peers, i.e., as long as the majority of privacy peers is honest. Even if some of the input peers do not trust each other, we think it is realistic to assume that they will agree on a set of most-trusted participants (or external entities) for hosting the privacy peers. Also, we think it is realistic to assume that the privacy peers indeed follow the protocol. If they are operated by input peers, they are likely interested in the correct outcome of the computation themselves and will therefore comply. External privacy peers are selected due to their good reputation or are being payed for a service. In both cases, they will do their best not to offend their customers by tricking the protocol.

The function $f()$ is specified *as if* a TTP was available. MPC guarantees that no information is leaked *from the computation process*. However, just learning the resulting value $f()$ could allow to infer sensitive information. For example, if the input bit of all input peers must remain secret, computing the logical AND of all input bits is insecure in itself: if the final result was 1, all input bits must be 1 as well and are thus no longer secret. *It is the responsibility of the input peers to verify that learning $f()$ is acceptable*, in the same way as they have to verify this when using a real TTP. For example, we assume input peers are not willing to reconstruct item distributions but consider it safe to compute the overall item count or entropy. To reduce the potential for deducing information from $f()$, protocols can enforce the submission of "valid" input data conforming to certain rules. For instance, in our event correlation protocol, the privacy peers verify that each input peer submits no duplicate events. More formally, the work on *differential privacy* [17] systematically randomizes the output $f()$ of database queries to prevent inference of sensitive input data.

Prior to running the protocols, the $m$ privacy peers set up a secure, i.e., confidential and authentic, channel to each other. In addition, each input peer creates a secure channel to each privacy peer. We assume that the required public keys and/or certificates have been securely distributed beforehand.

**Privacy-Performance Tradeoff** Although the number of privacy peers $m$ has a quadratic impact on the total communication and computation costs, there are also $m$ privacy peers sharing the load. That is, if the network capacity is sufficient, the overall running time of the protocols will scale linearly with $m$ rather than quadratically. On the other hand, the number of tolerated colluding privacy peers also scales linearly with $m$. Hence, the choice of $m$ involves a privacy-performance tradeoff. The separation of roles into input and privacy peers allows to tune this tradeoff independently of the number of input providers.

## 4.1 Event Correlation

The first protocol we present enables the input peers to privately aggregate arbitrary network events. An event $e$ is defined by a key-weight pair $e = (k, w)$. This notion is generic in the sense that keys can be defined to represent arbitrary types of network events, which are uniquely identifiable. The key $k$ could for instance be the source IP address of packets triggering IDS alerts, or the source address concatenated with a specific alert type or port number. It could also be the hash value of extracted malicious payload or represent a uniquely identifiable object, such as popular URLs, of which the input peers want to compute the total number of hits. The weight $w$ reflects the impact (count) of this event (object), e.g., the frequency of the event in the current time window or a classification on a severity scale.

Each input peer shares at most $s$ local events per time window. The goal of the protocol is to reconstruct an event if and only if a minimum number of input peers $T_c$ report the same event and the aggregated weight is at least $T_w$. The rationale behind this definition is that an input peer does not want to reconstruct local events that are unique in the set of all input peers, exposing sensitive information asymmetrically. But if the input peer knew that, for example, three other input peers report the same event, e.g., a specific intrusion alert, he would be willing to contribute his information and collaborate. Likewise, an input peer might only be interested in reconstructing events of a certain impact, having a non-negligible aggregated weight.

More formally, let $[e_{ij}] = ([k_{ij}], [w_{ij}])$ be the shared event $j$ of input peer $i$ with $j \leq s$ and $i \leq n$. Then we compute the aggregated count $C_{ij}$ and weight $W_{ij}$ according to (2) and (3) and reconstruct $e_{ij}$ iff (4) holds.

$$[C_{ij}] := \sum_{i' \neq i, j'} equal([k_{ij}], [k_{i'j'}]) \quad (2)$$

$$[W_{ij}] := \sum_{i' \neq i, j'} [w_{i'j'}] \cdot equal([k_{ij}], [k_{i'j'}]) \quad (3)$$

$$([C_{ij}] \geq T_c) \wedge ([W_{ij}] \geq T_w) \quad (4)$$

Reconstruction of an event $e_{ij}$ includes the reconstruction of $k_{ij}$, $C_{ij}$, $W_{ij}$, and the list of input peers reporting it, but the $w_{ij}$ remain secret. The detailed algorithm is given in Fig. 2.

**Input Verification** In addition to merely implementing the correlation logic, we devise two optional input verification steps. In particular, the PPs check that shared weights are below a maximum weight $w_{max}$ and that each input peer shares distinct events. These verifications are *not* needed to secure the computation process, but they serve two purposes. First, they protect from misconfigured input peers and flawed input data. Secondly, they protect against input peers that try to deduce information from the *final computation result*. For instance, an input peer could add an event $T_c - 1$ times (with a total weight of at least $T_w$) to find out whether any other input peers report the same event. These input verifications mitigate such attacks.

**Probe Response Attacks** If aggregated security events are made publicly available, this enables probe response attacks against the system [5]. The goal of probe response attacks is not to learn private input data but to identify the sensors of a distributed monitoring system. To remain undiscovered, attackers then exclude the known sensors from future attacks against the system. While defending against this in general is an intractable problem, [41] identified that the suppression of low-density attacks provides some protection against basic probe response attacks. Filtering out low-density attacks in our system can be achieved by setting the thresholds $T_c$ and $T_w$ sufficiently high.

**Complexity** The overall complexity, including verification steps, is summarized below in terms of operation invocations and rounds:

| | |
|---|---|
| $equal$: | $O\big((n - T_c)ns^2\big)$ |
| $lessThan$: | $(2n - T_c)s$ |
| $shortRange$: | $(n - T_c)s$ |
| multiplications: | $(n - T_c) \cdot (ns^2 + s)$ |
| rounds: | $7l + \log_2(n - T_c) + 26$ |

The protocol is clearly dominated by the number of $equal$ operations required for the aggregation step. It scales quadratically with $s$, however, depending on $T_c$, it scales linearly or quadratically with $n$. For instance, if $T_c$ has a constant offset to $n$ (e.g., $T_c = n - 4$), only $O(ns^2)$ $equal$s are required. However, if $T_c = n/2$, $O(n^2 s^2)$ $equal$s are necessary.

**Optimizations** To avoid the quadratic dependency on $s$, we are working on an MPC-version of a binary search algorithm that finds a secret $[a]$ in a sorted list of secrets $\{[b_1], \ldots, [b_s]\}$ with $\log_2 s$ comparisons by comparing $[a]$ to the element in the middle of the list, here called $[b_*]$. We then construct a new list, being the first or second half of the original list, depending on $lessThan([a], [b_*])$. The procedure is repeated recursively until the list has size 1. This allows us to compare all events of two input peers with only $O(s \log_2 s)$ instead of $O(s^2)$ comparisons. To further reduce the number of $equal$ operations, the protocol can be adapted to receive *incremental updates* from input peers. That is, input peers submit a list of events in each time window and inform the PPs, which event entries have a different key from the previous window. Then, only comparisons of updated keys have to be performed and overall complexity is reduced to $O(u(n - T_c)s)$, where $u$ is the number of changed keys in that window. This requires, of course, that information on input set dynamics is not considered private.

1. **Share Generation:** Each input peer $i$ shares $s$ distinct events $e_{ij}$ with $w_{ij} < w_{max}$ among the privacy peers (PPs).
2. **Weight Verification:** Optionally, the PPs compute and reconstruct $lessThan([w_{ij}], w_{max})$ for all weights to verify that they are smaller than $w_{max}$. Misbehaving input peers are disqualified.
3. **Key Verification:** Optionally, the PPs verify that each input peer $i$ reports distinct events, i.e., for each event index $a$ and $b$ with $a < b$ they compute and reconstruct $equal([k_{ia}], [k_{ib}])$. Misbehaving input peers are disqualified.
4. **Aggregation:** The PPs compute $[C_{ij}]$ and $[W_{ij}]$ according to (2) and (3) for $i \leq \hat{i}$ with $\hat{i} = min(n - T_c + 1, n)$. [2] All required $equal$ operations can be performed in parallel.
5. **Reconstruction:** For each event $[e_{ij}]$, with $i \leq \hat{i}$, condition (4) has to be checked. Therefore, the PPs compute

$$[t_1] = shortRange([C_{ij}], T_c, n), \qquad [t_2] = lessThan(T_w - 1, [W_{ij}])$$

Then, the event is reconstructed iff $[t_1] \cdot [t_2]$ returns 1. The set of input peers with $i > \hat{i}$ reporting a reconstructed event $\overline{r} = (\overline{k}, \overline{w})$ is computed by reusing all the $equal$ operations performed on $\overline{r}$ in the aggregation step. That is, input peer $i'$ reports $\overline{r}$ iff $\sum_j equal([\overline{k}], [k_{i'j}])$ equals 1. This can be computed using local addition for each remaining input peer and each reconstructed event. Finally, all reconstructed events are sent to all input peers.

Figure 2: Algorithm for event correlation protocol.

1. **Share Generation:** Each input peer $i$ shares its input vector $\mathbf{d_i} = (x_1, x_2, \ldots, x_r)$ among the PPs. That is, the PPs obtain $n$ vectors of sharings $[\mathbf{d_i}] = ([x_1], [x_2], \ldots, [x_r])$.
2. **Summation:** The PPs compute the sum $[\mathbf{D}] = \sum_{i=1}^{n} [\mathbf{d_i}]$.
3. **Reconstruction:** The PPs reconstruct all elements of $\mathbf{D}$ and send them to all input peers.

Figure 3: Algorithm for vector addition protocol.

1. **Share Generation:** Each input peer holds an $r$-dimensional private input vector $\mathbf{s^i} \in \mathbb{Z}_p^r$ representing the local item histogram, where $r$ is the number of items and $s_k^i$ is the count for item $k$. The input peers share all elements of their $\mathbf{s^i}$ among the PPs.
2. **Summation:** The PPs compute the item counts $[s_k] = \sum_{i=1}^{n} [s_k^i]$. Also, the total count $[S] = \sum_{k=1}^{r} [s_k]$ is computed and reconstructed.
3. **Exponentiation:** The PPs compute $[(s_k)^q]$ using square-and-multiply.
4. **Entropy Computation:** The PPs compute the sum $\sigma = \sum_k [(s_k)^q]$ and reconstruct $\sigma$. Finally, at least one PP uses $\sigma$ to (locally) compute the Tsallis entropy $H_q(Y) = \frac{1}{q-1}(1 - \sigma / S^q)$.

Figure 4: Algorithm for entropy protocol.

## 4.2 Network Traffic Statistics

In this section, we present protocols for the computation of multi-domain traffic statistics including the aggregation of additive traffic metrics, the computation of feature entropy, and the computation of distinct item count. These statistics find various applications in network monitoring and management.

### 4.2.1 Vector Addition

To support basic additive functionality on timeseries and histograms, we implement a vector addition protocol. Each input peer $i$ holds a private $r$-dimensional input vector $\mathbf{d_i} \in \mathbb{Z}_p^r$. Then, the vector addition protocol computes the sum $\mathbf{D} = \sum_{i=1}^{n} \mathbf{d_i}$. We describe the corresponding SEPIA protocol shortly in Fig. 3. This protocol requires no distributed multiplications and only one round.

### 4.2.2 Entropy Computation

The computation of the entropy of feature distributions has been successfully applied in network anomaly detection, e.g. [23, 9, 25, 50]. Commonly used feature distributions are, for example, those of IP addresses, port numbers, flow sizes or host degrees. The Shannon entropy of a feature distribution $Y$ is $H(Y) = -\sum_k p_k \cdot \log_2(p_k)$, where $p_k$ denotes the probability of an item $k$. If $Y$ is a distribution of port numbers, $p_k$ is the probability of

port $k$ to appear in the traffic data. The number of flows (or packets) containing item $k$ is divided by the overall flow (packet) count to calculate $p_k$. Tsallis entropy is a generalization of Shannon entropy that also finds applications in anomaly detection [50, 46]. It has been substantially studied with a rich bibliography available in [47]. The 1-parametric Tsallis entropy is defined as:

$$H_q(Y) = \frac{1}{q-1}\Big(1 - \sum_k (p_k)^q\Big). \qquad (5)$$

and has a direct interpretation in terms of moments of order $q$ of the distribution. In particular, the Tsallis entropy is a generalized, non-extensive entropy that, up to a multiplicative constant, equals the Shannon entropy for $q \to 1$. For generality, we select to design an MPC protocol for the Tsallis entropy.

**Entropy Protocol** A straight-forward approach to compute entropy is to first find the overall feature distribution $Y$ and then to compute the entropy of the distribution. In particular, let $p_k$ be the overall probability of item $k$ in the union of the private data and $s_k^i$ the local count of item $k$ at input peer $i$. If $S$ is the total count of the items, then $p_k = \frac{1}{S}\sum_{i=1}^{n} s_k^i$. Thus, to compute the entropy, the input peers could simply use the addition protocol to add all the $s_k^i$'s and find the probabilities $p_k$. Each input peer could then compute $H(Y)$ locally. However, the distribution $Y$ can still be very sensitive as it contains information for each item, e.g., per address prefix. For this reason, we aim at computing $H(Y)$ without reconstructing any of the values $s_k^i$ or $p_k$. Because the rational numbers $p_k$ can not be shared directly over a prime field, we perform the computation separately on private numerators ($s_k^i$) and the public overall item count $S$. The entropy protocol achieves this goal as described in Fig. 4. It is assured that sensitive intermediate results are not leaked and that input and privacy peers *only* learn the final entropy value $H_q(Y)$ and the total count $S$. $S$ is not considered sensitive as it only represents the total flow (or packet) count of all input peers together. This can be easily computed by applying the addition protocol to volume-based metrics. The complexity of this protocol is $r \log_2 q$ multiplications in $\log_2 q$ rounds.

#### 4.2.3 Distinct Count

In this section, we devise a simple distinct count protocol leaking no intermediate information. Let $s_k^i \in \{0, 1\}$ be a boolean variable equal to 1 if input peer $i$ sees item $k$ and 0 otherwise. We first compute the logical OR of the boolean variables to find if an item was seen by any input peer or not. Then, simply summing the number of variables equal to 1 gives the distinct count of the items. According to De Morgan's Theorem, $a \vee b = \neg(\neg a \wedge \neg b)$.

1. **Share Generation:** Each input peer $i$ shares its negated local counts $c_k^i = \neg s_k^i$ among the PPs.
2. **Aggregation:** For each item $k$, the PPs compute $[c_k] = [c_k^1] \wedge [c_k^2] \wedge \ldots [c_k^n]$. This can be done in $\log_2 n$ rounds. If an item $k$ is reported by any input peer, then $c_k$ is 0.
3. **Counting:** Finally, the PPs build the sum $[\sigma] = \sum[c_k]$ over all items and reconstruct $\sigma$. The distinct count is then given by $K - \sigma$, where $K$ is the size of the item domain.

Figure 5: Algorithm for distinct count protocol.

This means the logical OR can be realized by performing a logical AND on the negated variables. This is convenient, as the logical AND is simply the product of two variables. Using this observation, we construct the protocol described in Fig. 5. This protocol guarantees that only the distinct count is learned from the computation; the set of items is *not* reconstructed. However, if the input peers agree that the item set is not sensitive it can easily be reconstructed after step 2. The complexity of this protocol is $(n-1)r$ multiplications in $\log_2 n$ rounds.

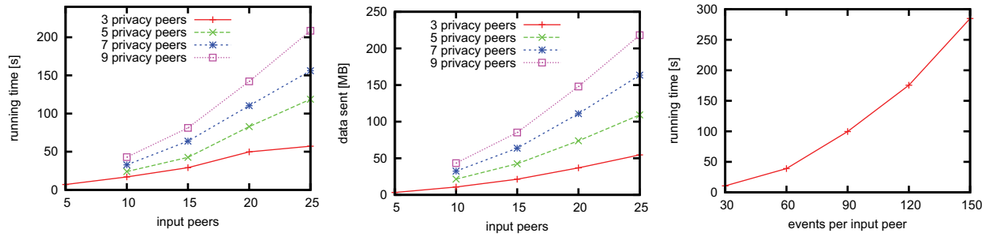
## 5 Performance Evaluation

In this Section we evaluate the event correlation protocol and the protocols for network statistics. After that we explore the impact of running selected protocols on PlanetLab where hardware, network delay, and bandwidth are very heterogeneous. This section is concluded with a performance comparison between SEPIA and existing general-purpose MPC frameworks.

We assessed the CPU and network bandwidth requirements of our protocols, by running different aggregation tasks with real and simulated network data. For each protocol, we ran several experiments varying the most important parameters. We varied the number of input peers $n$ between 5 and 25 and the number of privacy peers $m$ between 3 and 9, with $m < n$. The experiments were conducted on a shared cluster comprised of several public workstations; each workstation was equipped with a 2x Pentium 4 CPU (3.2 GHz), 2 GB memory, and 100 Mb/s network. Each input and privacy peer was run on a *separate* host. In our plots, each data point reflects the average over 10 time windows. Background load due to user activity could not be totally avoided. Section 5.3 discusses the impact of single slow hosts on the overall running time.

### 5.1 Event Correlation

For the evaluation of the event correlation protocol, we generated artificial event data. It is important to note that our performance metrics do not depend on the actual values used in the computation, hence artificial data is just as good as real data for these purposes.




(a) Average round time ($s = 30$). (b) Data sent per PP ($s = 30$). (c) Round time vs. $s$ ($n$=10, $m$=3).

Figure 6: Round statistics for event correlation with $T_c = n/2$. $s$ is the number of events per input peer.

**Running Time** Fig. 6 shows evaluation results for event correlation with $s = 30$ events per input peer, each with 24-bit keys for $T_c = n/2$. We ran the protocol including weight and key verification. Fig. 6a shows that the average running time per time window always stays below 3.5 min and scales quadratically with $n$, as expected. Investigation of CPU statistics shows that with increasing $n$ also the average CPU load per privacy peer grows. Thus, as long as CPUs are not used to capacity, local parallelization manages to compensate parts of the quadratical increase. With $T_c = n - const$, the running time as well as the number of operations scale linearly with $n$. Although the total communication cost grows quadratically with $m$, the running time dependence on $m$ is rather linear, as long as the network is not saturated. The dependence on the number of events per input peer $s$ is quadratic as expected without optimizations (see Fig. 6c).

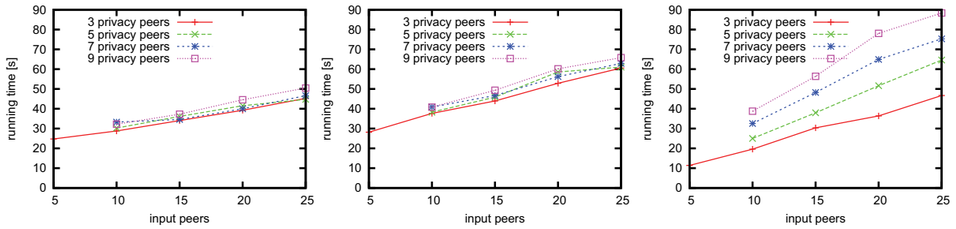
To study whether privacy peers spend most of their time waiting due to synchronization, we measured the user and system time of their hosts. All the privacy peers were constantly busy with average CPU loads between 120% and 200% for the various operations.[3] Communication and computation between PPs is implemented using separate threads to minimize the impact of synchronization on the overall running time. Thus, SEPIA profits from multi-core machines. Average load decreases with increasing need for synchronization from multiplications to $equal$, over $lessThan$ to event correlation. Nevertheless, even with event correlation, processors are very busy and not stalled by the network layer.

**Bandwidth requirements** Besides running time, the communication overhead imposed on the network is an important performance measure. Since data volume is dominated by privacy peer messages, we show the average bytes sent *per privacy peer* in one time window in Fig. 6b. Similar to running time, data volume scales roughly quadratically with $n$ and linearly with $m$. In addition to the transmitted data, each privacy peer receives about the same amount of data from the other input and private peers. If we assume a 5-minute clocking of the event correlation protocol, an average bandwidth between 0.4 Mbps (for $n = 5$, $m = 3$) and 13 Mbps (for $n = 25$, $m = 9$) is needed per privacy peer. Assuming a 5-minute interval and sufficient CPU/bandwidth resources, the maximum number of supported input peers before the system stops working in real-time ranges from around 30 up to roughly 100, depending on protocol parameters.

### 5.2 Network statistics

For evaluating the network statistics protocols, we used unsampled NetFlow data captured from the five border routers of the Swiss academic and research network (SWITCH), a medium-sized backbone operator, connecting approximately 40 governmental institutions, universities, and research labs to the Internet. We first extracted traffic flows belonging to different customers of SWITCH and assigned an independent input peer to each organization's trace. For each organization, we then generated SEPIA input files, where each input field contained either the values of volume metrics to be added or the local histogram of feature distributions for collaborative entropy (distinct count) calculation. In this section we focus on the running time and bandwidth requirements only. We performed the following tasks over ten 5-minute windows:

1. **Volume Metrics:** Adding 21 volume metrics containing flow, packet, and byte counts, both total and separately filtered by protocol (TCP, UDP, ICMP) and direction (incoming, outgoing). For example, Fig. 10 in Section 7.2 plots the total and local number of incoming UDP flows of six organizations for an 11-day period.




(a) Addition of port histogram. (b) Entropy of port distribution. (c) Distinct AS count.

Figure 7: Network statistics: avg. running time per time window versus $n$ and $m$, measured on a department-wide cluster. All tasks were run with an input set size of 65k items.

2. **Port Histogram:** Adding the full destination port histogram for incoming UDP flows. SEPIA input files contained 65,535 fields, each indicating the number of flows observed to the corresponding port. These local histograms were aggregated using the addition protocol.
3. **Port Entropy:** Computing the Tsallis entropy of destination ports for incoming UDP flows. The local SEPIA input files contained the same information as for histogram aggregation. The Tsallis exponent $q$ was set to 2.
4. **Distinct count of AS numbers:** Aggregating the count of distinct source AS numbers in incoming UDP traffic. The input files contained 65,535 columns, each denoting if the corresponding source AS number was observed. For this setting, we reduced the field size $p$ to 31 bits because the expected size of intermediate values is much smaller than for the other tasks.

**Running Time** For task 1, the average running time was below 1.6 s per time window for all configurations, even with 25 input and 9 privacy peers. This confirms that addition-only is very efficient for low volume input data. Fig. 7 summarizes the running time for tasks 2 to 4. The plots show on the $y$-axes the average running time per time window versus the number of input peers on the $x$-axes. In all cases, the running time for processing one time window was below 1.5 minutes. The running time clearly scales linearly with $n$. Assuming a 5-minute interval, we can estimate by extrapolation the maximum number of supported input peers before the system stops working in real-time. For the conservative case with 9 privacy peers, the supported number of input peers is approximately 140 for histogram addition, 110 for entropy computation, and 75 for distinct count computation. We observe, that for single round protocols (addition and entropy), the number of privacy peers has only little impact on the running time. For the distinct count protocol, the running time increases linearly with both $n$ and $m$. Note that the shortest running time for distinct count is even lower than for histogram addition. This is due to the reduced field size ($p$ with 31 bits instead of 62), which reduces both CPU and network load.

**Bandwidth Requirements** For all tasks, the data volume sent per privacy peer scales perfectly linear with $n$ and $m$. Therefore, we only report the maximum volume with 25 input and 9 privacy peers. For addition of volume metrics, the data volume is 141 KB and increases to 4.7 MB for histogram addition. Entropy computation requires 8.5 MB and finally the multi-round distinct count requires 50.5 MB. For distinct count, to transfer the total of $2 \cdot 50.5 = 101$ MB within 5 minutes, an average bandwidth of roughly 2.7 Mbps is needed per privacy peer.

## 5.3 Internet-wide Experiments

In our evaluation setting hosts have homogeneous CPUs, network bandwidth and low round trip times (RTT). In practice, however, SEPIA's goal is to aggregate traffic from remote network domains, possibly resulting in a much more heterogeneous setting. For instance, *high delay* and *low bandwidth* directly affect the waiting time for messages. Once data has arrived, the *CPU model and clock rate* determine how fast the data is processed and can be distributed for the next round.

Recall from Section 4 that each operation and protocol in SEPIA is designed in rounds. Communication and computation during each round run in parallel. But before the next round can start, the privacy peers have to synchronize intermediate results and therefore wait for the slowest privacy peer to finish. The overall running time of SEPIA protocols is thus affected by the slowest CPU, the highest delay, and the lowest bandwidth rather than by the average performance of hosts and links. Therefore we were interested to see whether the performance of our protocols breaks down if we take it out of the homogeneous LAN setting. Hence, we ran



| | LAN | PlanetLab A | PlanetLab B |
|---|---|---|---|
| Max. RTT | 1 ms | 320 ms | 320 ms |
| Bandwidth | 100 Mb/s | $\geq$ 100 Kb/s | $\geq$ 100 Kb/s |
| Slowest CPU | 2 cores | 2 cores | 1 core |
| | 3.2 GHz | 2.4 GHz | 1.8 GHz |
| Running time | 25.0 s | 36.8 s | 110.4 s |

Table 1: Comparison of LAN and PlanetLab settings.

| Framework | SEPIA | VIFF | FairplayMP |
|---|---|---|---|
| Technique | Shamir sh. | Shamir sh. | Bool. circuits |
| Platform | Java | Python | Java |
| Multipl./s | 82,730 | 326 | 1.6 |
| Equals/s | 2,070 | 2.4 | 2.3 |
| LessThans/s | 86 | 2.4 | 2.3 |

Table 2: Comparison of frameworks performance in operations per second with $m = 5$.

SEPIA on PlanetLab [31] and repeated task 4 (distinct AS count) with 10 input and 5 privacy peers on globally distributed PlanetLab nodes. Table 1 compares the LAN setup with two PlanetLab setups A and B.

RTT was much higher and average bandwidth much lower on PlanetLab. The only difference between PlanetLab A and B was the choice of some nodes with slower CPUs. Despite the very heterogeneous and globally distributed setting, the distinct count protocol performed well, at least in PlanetLab A. Most important, it still met our near real-time requirements. From PlanetLab A to B, running time went up by a factor of 3. However, this can largely be explained by the slower CPUs. The distinct count protocol consists of parallel multiplications, which make efficient use of the CPU and local addition, which is solely CPU-bound. Let us assume, for simplicity, that clock rates translate directly into MIPS. Then, computational power in PlanetLab B is roughly 2.7 times lower than in PlanetLab A. Of course, the more rounds a protocol has, the bigger is the impact of RTT. But in each round, the network delay is only a constant offset and can be amortized over the number of parallel operations performed per round. For many operations, CPU and bandwidth are the real bottlenecks.

While aggregation in a heterogeneous environment is possible, SEPIA *privacy peers* should ideally be deployed on dedicated hardware, to reduce background load, and with similar CPU equipment, so that no single host slows down the entire process.

## 5.4 Comparison with General-Purpose Frameworks

In this section we compare the performance of basic SEPIA operations to those of general-purpose frameworks such as FairplayMP [3] and VIFF v0.7.1 [15]. Besides performance, one aspect to consider is, of course, usability. Whereas the SEPIA library currently only provides an API to developers, FairplayMP allows to write protocols in a high-level language called SFDL and VIFF integrates nicely into the Python language. Furthermore, VIFF implements asynchronous protocols and provides additional functionality, such as security against malicious adversaries and support of MPC based on homomorphic cryptosystems.

Tests were run on 2x Dual Core AMD Opteron 275 machines with 1Gb/s LAN connections. To guarantee a fair comparison, we used the same settings for all frameworks. In particular, the semi-honest model, 5 computation nodes, and 32 bit secrets were used. Unlike VIFF and SEPIA, which use an information-theoretically secure scheme, FairplayMP requires the choice of an adequate security parameter $k$. We set $k = 80$, as suggested by the authors in [3].

Table 2 shows the average number of parallel operations per second for each framework. SEPIA clearly outperforms VIFF and FairplayMP for all operations and is thus much better suited when performance of parallel operations is of main importance. As an example, a run of event correlation taking 3 minutes with SEPIA would take roughly 2 days with VIFF. This extends the range of *practically* runnable MPC protocols significantly. Notably, SEPIA's $equal$ operation is 24 times faster than its $lessThan$, which requires 24 times more multiplications, but at the same time also twice the number of rounds. This confirms that with many parallel operations, the number of multiplications becomes the dominating factor. Approximately $3/4$ of the time spent for $lessThan$ is used for generating sharings of random numbers used in the protocol. These random sharings are independent from input data and could be generated prior to the actual computation, allowing to perform 380 $lessThan$s per second in the same setting.

Even for multiplications, SEPIA is faster than VIFF, although both rely on the same scheme. We assume this can largely be attributed to the completely asynchronous protocols implemented in VIFF. Whereas asynchronous protocols are very efficient for dealing with malicious adversaries, they make it impossible to reduce network overhead by exchanging intermediate results of all parallel operations at once in a single big message. Also, there seems to be a bottleneck in parallelizing large numbers of operations. In fact, when benchmarking VIFF, we noticed that after some point, adding more parallel operations significantly slowed down the average running time per operation.

Sharemind [6] is another interesting MPC framework using *additive* secret sharing to implement multiplications and greater-or-equal (GTE) comparison. The authors implement it in C++ to maximize performance. However, the use of additive secret sharing makes the im-

plementations of basic operations dependent on the number of computation nodes used. For this reason, Sharemind is currently restricted to 3 computation nodes only. Regarding performance, however, Sharemind is comparable to SEPIA. According to [6], Sharemind performs up to 160,000 multiplications and around 330 GTE operations per second, with 3 computation nodes. With 3 PPs, SEPIA performs around 145,000 multiplications and 145 $lessThan$s per second (615 with pre-generated randomness). Sharemind does not directly implement $equal$, but it could be implemented using 2 invocations of GTE, leading to $\approx$ 115 operations/s. SEPIA's $equal$ is clearly faster with up to $3,400$ invocations/s. SEPIA demonstrates that operations based on Shamir shares are not necessarily slower than operations in the additive sharing scheme. The key to performance is rather an implementation, which is optimized for a large number of parallel operations. Thus, SEPIA combines speed with the flexibility of Shamir shares, which support any number of computation nodes and are to a certain degree robust against node failures.

## 6 Design and Implementation

The foundation of the SEPIA library is an implementation of the basic operations, such as multiplications and optimized comparisons (see Section 3), along with a communication layer for establishing SSL connections between input and privacy peers. In order to limit the impact of varying communication latencies and response times, each connection, along with the corresponding computation and communication tasks, is handled by a separate thread. This also implies that SEPIA protocols benefit from multi-core systems for computation-intensive tasks. In order to reduce synchronization overhead, intermediate results of parallel operations sent to the same destination are collected and transfered in a single big message instead of many small messages. On top of the basic layers, the protocols from Section 4 are implemented as standalone command-line (CLI) tools. The CLI tools expect a local configuration file containing privacy peer addresses, paths to a folder with input data and a Java keystore, as well as protocol-dependent parameters. The tools write a log of the ongoing computation and output files with aggregate results for each time window. The keystore holds certificates of trusted input and privacy peers to establish SSL connections. It is possible to delay the start of a computation until a minimum number of input and privacy peers are online. This gives the input peers the ability to define an acceptable level of privacy by only participating in the computation if a certain number of other input/privacy peers also participate.

SEPIA is written in Java to provide platform independence. The source code of the basic library and the four

```
ShamirSharing sharing = new ShamirSharing();
sharing.setFieldPrime(1401085391); // 31 bit
sharing.setNrOfPrivacyPeers(nrOfPrivacyPeers);
sharing.init();

// Secret1: only a single value
long[] secrets = new long[]{1234567};
long[][] shares = sharing.generateShares(secrets);

// Send shares to each privacy peer
for(int i=0; i<nrOfPrivacyPeers; i++) {
  connection[i].sendMessage(shares[i]);
}
```

Figure 8: Example code for an input peer that shares a secret, e.g., a millionaire sharing his amount of wealth.

CLI tools is available under the LGPL license on the SEPIA project web page [39]. The web page also provides pre-configured examples for the CLI tools and a user manual. The user manual describes usage and configuration of the CLI tools and includes a step-by-step tutorial on how to use the library API to develop new protocols. In the library API, all operations and subprotocols implement a common interface `IOperation` and are easily composable. The class `ProtocolPrimitives` allows to schedule operations and takes care of performing them in parallel, keeping track of operation states. A base class for privacy peers implements the `doOperations()` method, which runs all the necessary computation rounds and synchronizes data between privacy peers in each round. Fig. 8 shows example code for input peers that want to privately compare their secrets. First, each input peer generates shares of its secret. The shares are then sent to the PPs, for which example code is shown in Fig. 9. The PPs first schedule and execute *lessThan* comparisons for all combinations of input secrets. In a second step, they run the reconstruction operations and output the results.

**Future Work** Note that with Shamir shares, reconstruction of results is assured as long as $t + 1$ PPs are online and responsive. This can be used directly to extend SEPIA protocols with robustness against node failures. Also, weak nodes slowing down the entire computation could be excluded from the computation. We leave this as a future extension.

The protocols support any number of input and privacy peers. Also, the item set sizes/events per input peer are configurable and thus only limited by the available CPU/bandwidth resources. However, running the network statistics protocols (e.g., distinct count) on very large distributions, such as the global IP address range, requires to use sketches as proposed in [37] or binning (e.g., use address prefixes instead of addresses). As an example, we have recently used sketches in combination with SEPIA to efficiently compute top-$k$ reports for dis-



```
... // receive all the shares from input peers
ProtocolPrimitives primitives = new ProtocolPrimitives(fieldPrime, ...);

// Schedule comparisons of all the input peer's secrets
int id1=1, id2=2, id3=3; // consecutive operation IDs
primitives.lessThan(id1, new long[]{shareOfSecret1, shareOfSecret2});
primitives.lessThan(id2, new long[]{shareOfSecret2, shareOfSecret3});
primitives.lessThan(id3, new long[]{shareOfSecret1, shareOfSecret3});
doOperations(); // Process operations and sychronize intermediate results

// Get shares of the comparison results
long shareOfLessThan12 = primitives.getResult(id1);
long shareOfLessThan23 = primitives.getResult(id2);
long shareOfLessThan13 = primitives.getResult(id3);

// Schedule and perform reconstruction of comparisons
primitives.reconstruct(id1, new long[]{shareOfLessThan12});
primitives.reconstruct(id2, new long[]{shareOfLessThan23});
primitives.reconstruct(id3, new long[]{shareOfLessThan13});
doOperations();

boolean secret1_lessThan_secret2 = (primitives.getResult(id1)==1);
boolean secret2_lessThan_secret3 = (primitives.getResult(id2)==1);
boolean secret1_lessThan_secret3 = (primitives.getResult(id3)==1);
```

Figure 9: Example code for a PP receiving shares of secrets from 3 input peers. It then compares the secrets privately, e.g., to find which of the millionaires is the richest.

tributed IP address distributions with up to 180,000 distinct addresses [10].

As part of future work, we also plan to investigate the applicability of polynomial set representation to our statistics protocols, to reduce the linear dependency on the input set domain. Polynomial set representation, introduced by Freedman *et al.* [18] and extended by Kissner *et al.* [22], represents set elements as roots of a polynomial and enables set operations that scale only logarithmically with input domain size. However, these solutions use homomorphic public-key cryptosystems, which come with significant overhead for basic operations. Furthermore, they do not trivially allow to separate roles into input and privacy peers, as each input provider is required to perform certain non-delegable processing steps on its own data.

## 7 Applications

We envision four distinct aggregation scenarios using SEPIA. The first scenario is aggregating information coming from multiple domains of one large (international) organization. This aggregation is presently not always possible due to privacy concerns and heterogeneous jurisdiction. The second scenario is analyzing private data owned by independent organizations with a mutual benefit in collaborating. Local ISPs, for example, might collaborate to detect common attacks. A third scenario provides access to researchers for evaluating and validating traffic analysis or event correlation prototypes over multi-domain network data. For example, national research, educational, and university networks could provide SEPIA input and/or privacy peers that allow analyzing local data according to submitted MPC scripts. Finally, one last scenario is the privacy-preserving analysis of end-user data, i.e., end-user workstations can use SEPIA to collaboratively analyze and cross-correlate local data.

### 7.1 Application Taxonomy

Based on these scenarios, we see three different classes of possible SEPIA applications.

**Network Security** Over the last years, considerable research efforts have focused on distributed data aggregation and correlation systems for the identification and mitigation of coordinated wide-scale attacks. In particular, aggregation enables the (early) detection and characterization of attacks spanning multiple domains using data from IDSes, firewalls, and other possible sources [2, 16, 26, 49]. Recent studies [21] show that coordinated wide-scale attacks are prevalent: 20% of the studied malicious addresses and 40% of the IDS alerts accounted for coordinated wide-scale attacks. Furthermore, strongly correlated groups profiting most from collaboration have less than 10 members and are stable over time, which is well-suited for SEPIA protocols.

In order to counter such attacks, Yegneswaran *et al.* [49] presented DOMINO, a distributed IDS that enables collaboration among nodes. They evaluated the performance of DOMINO with a large set of IDS logs from over 1600 providers. Their analysis demonstrates the significant benefit that is obtained by correlating the data from several distributed intrusion data sources. The major issue faced by such correlation systems is the lack

of data privacy. In their work, Porras *et al.* survey existing defense mechanisms and propose several remaining research challenges [32]. Specifically, they point out the need for efficient privacy-preserving data mining algorithms that enable traffic classification, signature extraction, and propagation analysis.

**Profiling and Performance Analysis** A second category of applications relates to traffic profiling and performance measurements. A global profile of traffic trends helps organizations to cross-correlate local traffic trends and identify changes. In [38] the authors estimate that 50 of the top-degree ASes together cover approximately 90% of global AS-paths. Hence, if large ASes collaborate, the computation of global Internet statistics is within reach. One possible statistic is the total traffic volume across a large number of networks. This statistic, for example, could have helped [37] in the dot-com bubble in the late nineties, since the traffic growth rate was overestimated by a factor of 10, easing the flow of venture capital to Internet start-ups. In addition, performance-related applications can benefit from an "on average" view across multiple domains. Data from multiple domains can also help to locate a remote outage with higher confidence, and to trigger proper detour mechanisms. A number of additional MPC applications related to performance monitoring are discussed in [36].

**Research Validation** Many studies are obliged to avoid rigorous validation or have to re-use a small number of old traffic traces [13, 43]. This situation clearly undermines the reliability of the derived results. In this context, SEPIA can be used to establish a privacy-preserving infrastructure for research validation purposes. For example, researchers could provide MPC scripts to SEPIA nodes running at universities and research institutes.

## 7.2 Case Study: The Skype Outage

The Skype outage in August 2007 started from a Windows update triggering a large number of system restarts. In response, Skype nodes scanned cached host-lists to find supernodes causing a huge distributed scanning event lasting two days [35]. We used NetFlow traces of the actual up- and downstream traffic of the 17 biggest customers of the SWITCH network. The traces span 11 days from the 11th to 22nd and include the Skype outage (on the 16th/17th) along with other smaller anomalies. We ran SEPIA's total count, distinct count, and entropy protocols on these traces and investigated how the organizations can benefit by correlating their local view with the aggregate view.

We first computed per-organization and aggregate timeseries of the UDP flow count metric and applied a simple detector to identify anomalies. For each time-

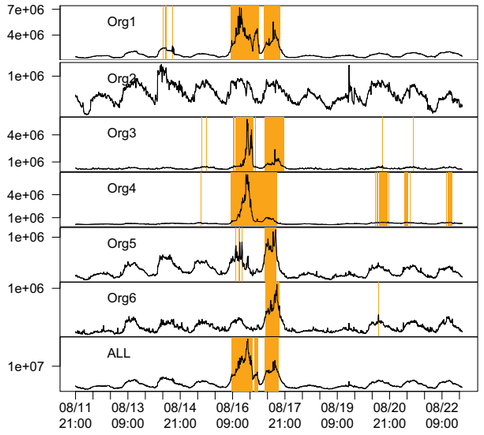



Figure 10: Flow count in 5' windows with anomalies for the biggest organizations and aggregate view (ALL). Each organization sees its local and the aggregate traffic.

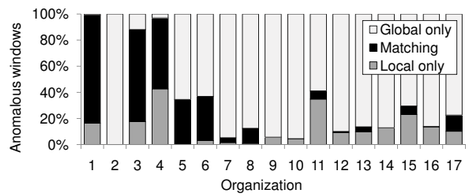



Figure 11: Correlation of local and global anomalies for organizations ordered by size (1=biggest).

series, we used the first 4 days to learn its mean $\mu$ and standard deviation $\sigma$, defined the normal region to be within $\mu \pm 3\sigma$, and detected anomalous time intervals. In Fig. 10 we illustrate the local timeseries for the six largest organizations and the aggregate timeseries. We rank organizations based on their decreasing average number of daily flows and use their rank to identify them. In the figure, we also mark the detected anomalous intervals. Observe that in addition to the Skype outage, some organizations detect other smaller anomalies that took place during the 11-day period.

**Anomaly Correlation** Using the aggregate view, an organization can find if a local anomaly is the result of a global event that may affect multiple organizations. Knowing the global or local nature of an anomaly is important for steering further troubleshooting steps. Therefore, we first investigate how the local and global anomalous intervals correlate. For each organization, we compared the local and aggregate anomalous intervals and measured the total time an anomaly was present: 1) only



in the local view, 2) only in the aggregate view, and 3) both in the local and aggregate views, i.e., the *matching anomalous intervals*. Fig. 11 illustrates the corresponding time fractions. We observe a rather small fraction, i.e., on average 14.1%, of local-only anomalies. Such anomalies lead administrators to search for local targeted attacks, misconfigured or compromised internal systems, misbehaving users, etc. In addition, we observe an average of 20.3% matching anomalous windows. Knowing an anomaly is both local and global steers an affected organization to search for possible problems in popular services, in widely-used software, like Skype in this case, or in the upstream providers. A large fraction (65.6%) of anomalous windows is only visible in the global view. In addition, we observe significant variability in the patterns of different organizations. In general, larger organizations tend to have a larger fraction of matching anomalies, as they contribute more to the aggregate view. While some organizations are highly correlated with the global view, e.g., organization 3 that notably contributes only 7.4% of the total traffic; other organizations are barely correlated, e.g., organizations 9 and 12; and organization 2 has no local anomalies at all.

**Anomaly Troubleshooting** We define *relative anomaly size* to be the ratio of the detection metric value during an anomalous interval over the detection threshold. Organizations 3 and 4 had relative anomaly sizes 11.7 and 18.8, which is significantly higher than the average of 2.6. Using the average statistic, organizations can compare the relative impact of an attack. Organization 2, for instance, had anomaly size 0 and concludes that there was a large anomaly taking place but they were not affected. Most of the organizations conclude that they were indeed affected, but less than average. Organizations 3 and 4, however, have to spend thoughts on why the anomaly was so disproportionately strong in their networks.

An investigation of the full port distribution and its entropy (plots omitted due to space limitations) shows that affected organizations see a sudden increase in scanning activity on specific high port numbers. Connections originate mainly from ports 80 and 443, i.e., the fallback ports of Skype, and a series of high port numbers indicating an anomaly related to Skype. For organizations 3 and 4, some of the scanned high ports are extremely prevalent, i.e., a single destination port accounts for 93% of all flows at the peak rate. Moreover, most of the anomalous flows within organizations 3 and 4 are targeted at a single IP address and originate from thousands of distinct source addresses connecting repeatedly up to 13 times per minute. These patterns indicate that the two organizations host popular supernodes, attracting a lot of traffic to specific ports. Other organizations mainly host client nodes and see uniform scanning, while organiza-

| Org # | 3 | 5 | 6 | 7 | 13 | 17 |
|---|---|---|---|---|---|---|
| lag [hours] | 1.2 | 2.7 | 23.4 | 15.5 | 4.8 | 3.6 |

Table 3: Organizations profiting from an early anomaly warning by aggregation.

tion 2 has banned Skype completely. Based on this analysis, organizations can take appropriate measures to mitigate the impact of the 2-day outage, like notifying users or blocking specific port numbers.

**Early-warning** Finally, we investigate whether the aggregate view can be useful for building an early-warning system for global or large-scale anomalies. The Skype anomaly did not start concurrently in all locations, since the Windows update policy and reboot times were different across organizations. We measured the lag between the time the Skype anomaly was first observed in the aggregate and local view of each organization. In Table 3 we list the organizations that had considerable lag, i.e., above an hour. Notably, one of the most affected organizations (6) could have learned the anomaly almost one day ahead. However, as shown in Fig. 11, for organization 2 this would have been a false positive alarm. To profit most from such an early warning system in practice, the aggregate view should be annotated with additional information, such as the number of organizations or the type of services affected from the same anomaly. In this context, our event correlation protocol is useful to decide whether similar anomaly signatures are observed in the participating networks. Anomaly signatures can be extracted automatically using actively researched techniques [8, 33].

# 8 Related Work

Most related to our work, Roughan and Zhan [37] first proposed the use of MPC techniques for a number of applications relating to traffic measurements, including the estimation of global traffic volume and performance measurements [36]. In addition, the authors identified that MPC techniques can be combined with commonly-used traffic analysis methods and tools, such as time-series algorithms and sketch data structures. Our work is similar in spirit, yet it extends their work by introducing new MPC protocols for event correlation, entropy, and distinct count computation and by implementing these protocols in a ready-to-use library.

Data correlation systems that provide strong privacy guarantees for the participants achieve data privacy by means of (partial) data sanitization based on bloom filters [44] or cryptographic functions [26, 24]. However, data sanitization is in general not a lossless process and

therefore imposes an unavoidable tradeoff between data privacy and data utility.

The work presented by Chow *et al.* [12] and Applebaum *et al.* [1] avoid this tradeoff by means of cryptographic data obfuscation. Chow *et al.* proposed a two-party query computation model to perform privacy-preserving querying of distributed databases. In addition to the databases, their solution comprises three entities: the randomizer, the computing engine, and the query frontend. Local answers to queries are randomized by each database and the aggregate results are derandomized at the frontend. Applebaum *et al.* present a semi-centralized solution for the collaboration among a large number of participants in which responsibility is divided between a proxy and a central database. In a first step the proxy obliviously blinds the clients' input, consisting of a set of keyword/value pairs, and stores the blinded keywords along with the non-blinded values in the central database. On request, the database identifies the (blinded) keywords that have values satisfying some evaluation function and forwards the matching rows to the proxy, which then unblinds the respective keywords. Finally, the database publishes its non-blinded data for these keywords. As opposed to these approaches, SEPIA does not depend on two central entities but in general supports an arbitrary number of distributed privacy peers, is provably secure, and more flexible with respect to the functions that can be executed on the input data. The similarities and differences between our work and existing general-purpose MPC frameworks are discussed in Sec. 5.4.

## 9 Conclusion

The aggregation of network security and monitoring data is crucial for a wide variety of tasks, including collaborative network defense and cross-sectional Internet monitoring. Unfortunately, concerns regarding privacy prevent such collaboration from materializing. In this paper, we investigated the practical usefulness of solutions based on secure multiparty computation (MPC). For this purpose, we designed optimized MPC operations that run efficiently on voluminous input data. We implemented these operations in the SEPIA library along with a set of novel protocols for event correlation and for computing multi-domain network statistics, i.e., entropy and distinct count. Our evaluation results clearly demonstrate the efficiency and scalability of SEPIA in realistic settings. With COTS hardware, near real-time operation is practical even with 140 input providers and 9 computation nodes. Furthermore, the basic operations of the SEPIA library are significantly faster than those of existing MPC frameworks and can be used as building blocks for arbitrary protocols. We believe that our work provides useful insights into the practical utility of MPC and paves the way for new collaboration initiatives. Our future work includes improving SEPIA's robustness against host failures, dealing with malicious adversaries, and further improving performance, using, for example, polynomial set representations. Furthermore, in collaboration with a major systems management vendor, we have started a project that aims at incorporating MPC primitives into a mainstream traffic profiling product.

## Acknowledgments

We are grateful to SWITCH for providing their traffic traces and to the anonymous reviewers for their helpful comments. Also, we want to thank Lisa Barisic and Dominik Schatzmann for their contributions. Special thanks go to Vassilis Zikas for assisting with MPC matters and for the idea of using Fermat's little theorem.

## References

[1] Applebaum, B., Ringberg, H., Freedman, M. J., Caesar, M., and Rexford, J. Collaborative, privacy-preserving data aggregation at scale. In *Privacy Enhancing Technologies Symposium (PETS)* (2010).

[2] ATLAS. Active Threat Level Analysis System. `http://atlas.arbor.net`.

[3] Ben-David, A., Nisan, N., and Pinkas, B. FairplayMP: a system for secure multi-party computation. In *Conference on Computer and communications security (CCS)* (2008).

[4] Ben-Or, M., Goldwasser, S., and Wigderson, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *ACM symposium on Theory of computing (STOC)* (1988).

[5] Bethencourt, J., Franklin, J., and Vernon, M. Mapping internet sensors with probe response attacks. In *14th USENIX Security Symposium* (2005).

[6] Bogdanov, D., Laur, S., and Willemson, J. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *European Symposium on Research in Computer Security (ESORICS)* (2008).

[7] Bogetoft, P., Christensen, D., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J., Nielsen, J., Nielsen, K., Pagter, J., et al. Secure multiparty computation goes live. In *Financial Cryptography* (2009).

[8] Brauckhoff, D., Dimitropoulos, X., Wagner, A., and Salamatian, K. Anomaly extraction in backbone networks using association rules. In *ACM SIGCOMM/USENIX Internet Measurement Conference (IMC)* (2009).

[9] Brauckhoff, D., Salamatian, K., and May, M. Applying PCA for Traffic Anomaly Detection: Problems and Solutions. In *INFOCOM* (2009).

[10] Burkhart, M., and Dimitropoulos, X. Fast privacy-preserving top-k queries using secret sharing. In *International Conference on Computer Communication Networks (ICCCN)* (2010).

[11] Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science (FOCS)* (2001).

[12] Chow, S. S. M., Lee, J.-H., and Subramanian, L. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS* (2009), The Internet Society.



[13] Claffy, K., Crovella, M., Friedman, T., Shannon, C., and Spring, N. Community-Oriented Network Measurement Infrastructure (CONMI) Workshop Report. *Computer Communication Review (CCR) 36*, 2 (2006), 41.

[14] Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J., and Toft, T. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference (TCC)* (2006).

[15] Damgård, I., Geisler, M., Krøigaard, M., and Nielsen, J. Asynchronous multiparty computation: Theory and implementation. In *Conference on Practice and Theory in Public Key Cryptography (PKC)* (2009).

[16] DShield. The Internet Storm Center. `www.dshield.org`.

[17] Dwork, C. Differential privacy: A survey of results. *Theory and Applications of Models of Computation (TAMC)* (2008).

[18] Freedman, M. J., Nissim, K., and Pinkas, B. Efficient Private Matching and Set Intersection. In *EUROCRYPT '04* (2004).

[19] Gennaro, R., Rabin, M., and Rabin, T. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *7th annual ACM symposium on Principles of distributed computing (PODC)* (1998).

[20] Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game. In *ACM symposium on Theory of computing (STOC)* (1987).

[21] Katti, S., Krishnamurthy, B., and Katabi, D. Collaborating against common enemies. In *ACM SIGCOMM/USENIX Internet Measurement Conference (IMC)* (2005).

[22] Kissner, L., and Song, D. Privacy-Preserving Set Operations. In *Proceedings of CRYPTO '05* (2005).

[23] Lakhina, A., Crovella, M., and Diot, C. Mining anomalies using traffic feature distributions. In *ACM SIGCOMM* (2005).

[24] Lee, A. J., Tabriz, P., and Borisov, N. A privacy-preserving interdomain audit framework. In *Workshop on privacy in electronic society (WPES)* (2006).

[25] Li, X., Bian, F., Crovella, M., Diot, C., Govindan, R., Iannaccone, G., and Lakhina, A. Detection and identification of network anomalies using sketch subspaces. In *ACM SIGCOMM/USENIX Internet Measurement Conference (IMC)* (2006).

[26] Lincoln, P., Porras, P., and Shmatikov, V. Privacy-preserving sharing and correlation of security alerts. In *13th USENIX Security Symposium* (2004).

[27] Machiraju, S., and Katz, R. H. Verifying global invariants in multi-provider distributed systems. In *SIGCOMM Workshop on Hot Topics in Networking (HotNets)* (2004), ACM.

[28] Nishide, T., and Ohta, K. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Conference on Theory and Practice of Public Key Cryptography (PKC)* (2007).

[29] Ohm, P. Broken promises of privacy: Responding to the surprising failure of anonymization. *57 UCLA Law Review* (2010). Available at `http://ssrn.com/abstract=1450006`.

[30] Parekh, J. J., Wang, K., and Stolfo, S. J. Privacy-preserving payload-based correlation for accurate malicious traffic detection. In *ACM Workshop on Large-scale Attack Defense (LSAD)* (2006).

[31] PlanetLab. An open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org`.

[32] Porras, P., and Shmatikov, V. Large-scale collection and sanitization of network security data: risks and challenges. In *Workshop on New security paradigms (NSPW)* (2006).

[33] Ranjan, S., Shah, S., Nucci, A., Munafò, M. M., Cruz, R. L., and Muthukrishnan, S. M. Dowitcher: Effective worm detection and containment in the internet core. In *INFOCOM* (2007).

[34] Ringberg, H. *Privacy-Preserving Collaborative Anomaly Detection*. PhD thesis, Princeton University, 2009.

[35] Rossi, D., Mellia, M., and Meo, M. Understanding Skype Signaling. *Computer Networks 53*, 2 (2009), 130–140.

[36] Roughan, M., and Zhang, Y. Privacy-preserving performance measurements. In *SIGCOMM workshop on Mining network data (MineNet)* (2006).

[37] Roughan, M., and Zhang, Y. Secure distributed data-mining and its application to large-scale network measurements. *Computer Communication Review (CCR) 36*, 1 (2006), 7–14.

[38] Sekar, V., Xie, Y., Maltz, D., Reiter, M., and Zhang, H. Toward a framework for internet forensic analysis. In *ACM HotNets-III* (2004).

[39] SEPIA web page. `http://www.sepia.ee.ethz.ch`.

[40] Shamir, A. How to share a secret. *Communications of the ACM 22*, 11 (1979), 612–613.

[41] Shmatikov, V., and Wang, M. Security against probe-response attacks in collaborative intrusion detection. In *ACM Workshop on Large-scale Attack Defense (LSAD)* (2007).

[42] Simpson, C. R., Jr., and Riley, G. F. Neti@home: A distributed approach to collecting end-to-end network performance measurements. In *Passive and Active Measurement Conference (PAM)* (2004).

[43] Slagell, A., and Yurcik, W. Sharing Computer Network Logs for Security and Privacy: A Motivation for New Methodologies of Anonymization. In *Workshop on the Value of Security through Collaboration (SECOVAL)* (September 2005).

[44] Stolfo, S. J. Worm and attack early warning. *IEEE Security and Privacy 2*, 3 (2004), 73–75.

[45] Tariq, M. B., Motiwala, M., Feamster, N., and Ammar, M. Detecting network neutrality violations with causal inference. In *Conference on Emerging networking experiments and technologies (CoNEXT)* (2009).

[46] Tellenbach, B., Burkhart, M., Sornette, D., and Maillart, T. Beyond Shannon: Characterizing Internet Traffic with Generalized Entropy Metrics. In *Passive and Active Measurement Conference (PAM)* (April 2009).

[47] Nonextensive statistical mechanics and thermodynamics. `http://tsallis.cat.cbpf.br/biblio.htm`.

[48] Yao, A. Protocols for secure computations. In *IEEE Symposium on Foundations of Computer Science* (1982).

[49] Yegneswaran, V., Barford, P., and Jha, S. Global Intrusion Detection in the DOMINO Overlay System. In *Network and Distributed System Security Symposium (NDSS)* (2004).

[50] Ziviani, A., Gomes, A., Monsores, M., and Rodrigues, P. Network anomaly detection using nonextensive entropy. *Communications Letters, IEEE 11*, 12 (2007), 1034–1036.

## Notes

[1] We define *near real-time* as the requirement of fully processing an $x$-minute interval of traffic data in no longer than $x$ minutes, where $x$ is typically a small constant. For our evaluation, we use 5-minute windows.

[2] For instance, if $n = 10$ and $T_c = 7$, each event that needs to be reconstructed according to (4) must be reported by at least one of the first 4 input peers. Hence, it is sufficient to compute the $C_{ij}$ and $W_{ij}$ for the first $n - T_c + 1 = 4$ input peers.

[3] When run on a 32-bit platform, up to twice the CPU load was observed, with similar overall running time. This difference is due to shares being stored in `long` variables, which are more efficiently processed on 64-bit CPUs.

# Dude, where's that IP? Circumventing measurement-based IP geolocation

*Phillipa Gill* *Yashar Ganjali*
*Dept. of Computer Science*
*University of Toronto*

*Bernard Wong*
*Dept. of Computer Science*
*Cornell University*

*David Lie*
*Dept. of Electrical and Computer Engineering*
*University of Toronto*

## Abstract

Many applications of IP geolocation can benefit from geolocation that is robust to adversarial clients. These include applications that limit access to online content to a specific geographic region and cloud computing, where some organizations must ensure their virtual machines stay in an appropriate geographic region. This paper studies the applicability of current IP geolocation techniques against an adversary who tries to subvert the techniques into returning a forged result. We propose and evaluate attacks on both delay-based IP geolocation techniques and more advanced topology-aware techniques. Against delay-based techniques, we find that the adversary has a clear trade-off between the accuracy and the detectability of an attack. In contrast, we observe that more sophisticated topology-aware techniques actually fare worse against an adversary because they give the adversary more inputs to manipulate through their use of topology and delay information.

## 1 Introduction

Many applications benefit from using IP geolocation to determine the geographic location of hosts on the Internet. For example, online advertisers and search engines tailor their content based on the client's location. Currently, geolocation databases such as Quova [22] and MaxMind [16] are the most popular method used by applications that need geolocation services.

Geolocation is also used in many security-sensitive applications. Online content providers such as Hulu [13], BBC iPlayer [22], RealMedia [22] and Pandora [20], limit their content distribution to specific geographic regions. Before allowing a client to view the content, they determine the client's location from its IP address and allow access only if the client is in a permitted jurisdiction. In addition, Internet gambling websites must restrict access to their applications based on the client's location or risk legal repercussions [29]. Accordingly, these businesses rely on geolocation to limit access to their online services.

Looking forward, the growth of infrastructure-as-a-service clouds, such as Amazon's EC2 service [1], may also drive organizations using cloud computing to employ geolocation. Users of cloud computing deploy VMs on a cloud provider's infrastructure without having to maintain the hardware their VM is running on. However, differences in laws governing issues such as privacy, information discovery, compliance and audit require that some cloud users to restrict VM locations to certain jurisdictions or countries [6]. These location restrictions may be specified as part of a service level agreement (SLA) between the cloud user and provider. Cloud users can use IP geolocation to independently verify that the location restrictions in their cloud SLAs are met.

In these cases, the target of geolocation has an incentive to mislead the geolocation system about its true location. Clients commonly use proxies to mislead content providers so they can view content that is unauthorized in their geographic region. In response, some content providers [13] however, have identified and blocked access from known proxies; but this does not prevent all clients from circumventing geographic controls. Similarly, cloud providers may attempt to break location restrictions in their SLAs to move customer VMs to cheaper locations. Governments that enforce location requirements on the cloud user may require the geolocation checks to be robust *no matter what* a cloud provider may do to mislead them. Even if the cloud provider itself is not malicious, its employees may also try to relocate VMs to locations where they can be attacked by other malicious VMs [24]. Thus, while cloud users might trust the cloud service provider, they may still be required to cd ..have independent verification of the location of their VMs to meet audit requirements or to avoid legal liability.

IP geolocation has been an active field of research for almost a decade. However, all current geolocation techniques assume a benign target that is not trying to intentionally mislead the user, and there has been limited work on geolocating malicious targets. Castelluccia *et al.* apply Constraint-Based Geolocation (CBG) [12] to the problem of geolocating fast-flux hidden servers that use a layer of proxies in a botnet [5] to conceal their location. Muir and Oorschot [18] describe limitations of passive geolocation techniques (e.g., `whois` services) and present a technique for finding the IP address of a machine using the Tor anonymization network [28]. These previous works focus on de-anonymization of hosts behind proxies, while our contribution in this paper is to answer fundamental questions about whether current geolocation algorithms are suitable for security-sensitive applications:

- **Are current geolocation algorithms accurate enough to locate an IP within a certain country or jurisdiction?** We answer this question by surveying previously published studies of geolocation algorithms. We find that current algorithms have accuracies of 35-194 km, making them suitable for geolocation within a country.
- **How can adversaries attack a geolocation system?** We propose attacks on two broad classes of measurement-based geolocation algorithms – those relying on network delay measurements and those using network topology information. To evaluate the practicality of these attacks, we categorize adversaries into two classes – a simple adversary that can manipulate network delays and a sophisticated one with control over a set of routable IP addresses.
- **How effective are such attacks? Can they be detected?** We evaluate our attacks by analyzing them against models of geolocation algorithms. We also perform an empirical evaluation using measurements taken from PlanetLab [21] and executing attacks on implementations of delay-based and topology-aware geolocation algorithms. We observe the simple adversary has limited accuracy and must trade off accuracy for detectability of their attack. On the other hand, the sophisticated adversary has higher accuracy and remains difficult to detect.

The rest of this paper is structured as follows. Section 2 summarizes relevant background and previous work on geolocation techniques. The security model and assumptions we use to evaluate current geolocation proposals is described in Section 3. We develop and analyze attacks on delay-based and topology-aware geolocation methods in Sections 4 and 5, respectively. Section 6 presents related work that evaluates geolocation when confronted by a target that leverages proxies. We present conclusions in Section 7.

## 2 Geolocation Background

IP geolocation aims to solve the problem of determining the geographic location of a given IP address. The solution can be expressed to varying degrees of granularity; for most applications the result should be precise enough to determine the city in which the IP is located, either returning a city name or the longitude and latitude where the target is located. The two main approaches to geolocation use either active network measurements to determine the location of the host or databases of IP to location mappings.

Measurement-based geolocation algorithms [9, 12, 14, 19, 30, 31] leverage a set of geographically distributed *landmark* hosts with known locations to locate the *target* IP. These landmarks measure various network properties, such as delay, and the paths taken by traffic between themselves and the target. These results are used as input to the geolocation algorithm which uses them to determine the target's location using methods such as: constraining the region where the target may be located (geolocalization) [12, 30], iterative force directed algorithms [31], machine learning [9] and constrained optimization [14].

Geolocation algorithms mainly rely on `ping` [7] and `traceroute` [7] measurements. Ping measures the round-trip time (RTT) delay between two machines on the Internet, while `traceroute` discovers and measures the RTT to routers along the path to a given destination. We classify measurement-based geolocation algorithms by the type of measurements they use to determine the target's location. We refer to algorithms that use end-to-end RTTs as delay-based [9, 12, 31] and those that use both RTT and topology information as topology-aware algorithms [14, 30].

An alternative to measurement-based geolocation is geolocation using databases of IP to location mappings. These databases can be either proprietary or public. Public databases include those administered by regional Internet registries (e.g., ARIN [3], RIPE [23]). Proprietary databases of IP to geographic location mappings are provided by companies such as Quova [22] and Maxmind [16]. While the exact method of constructing these databases is not public, they are sometimes based on a combination of `whois` services, DNS LOC records and autonomous system (AS) numbers [2]. Registries and databases tend to be coarse grained, usually returning the headquarters location of the organization that registered the IP address. This becomes a problem when organizations distribute their IP addresses over a wide geographic region, such as large ISPs or content providers. Misleading database geolocation is also straightforward through the use of proxies.



Table 1: Average accuracy of measurement-based geolocation algorithms.

| Class | Algorithm | Average accuracy (km) |
|---|---|---|
| Delay-based | GeoPing [19] | 150 km (25th percentile); 109 km (median) [30] |
| | CBG [12] | 78-182 |
| | Statistical [31] | 92 |
| | Learning-based [9] | 407-449 (113 km less than CBG [12] on their data) |
| Topology-aware | TBG [14] | 194 |
| | Octant [30] | 35-40 (median) |
| Other | GeoTrack [19] | 156 km (median) [30] |

DNS LOC [8] is an open standard that allows DNS administrators to augment DNS servers with location information, effectively creating a publicly available database of IP location information. However, it has not gained widespread usage. In addition, since the contents of the DNS LOC database are not authenticated and are set by the owners of the IP addresses themselves, it is poorly suited for security-sensitive applications.

Much research has gone into improving the accuracy of measurement-based geolocation algorithms; consequently, they provide fairly reliable results. Table 1 shows the reported average accuracies of recently proposed geolocation algorithms. Based on the reported accuracies, we believe that current geolocation algorithms are sufficiently accurate to place a machine within a country or jurisdiction. In particular, CBG [12] and Octant [30] appear to offer accuracies well within the size of most countries and may even be able to place users within a metropolitan area. Measurement-based geolocation is particularly appealing for secure geolocation because if a measurement can reach the target (e.g., using application layer measurements [17]), even if it is behind a proxy (e.g., SOCKS or HTTP proxy), the effectiveness of proxying will be diminished.

## 3 Security Model

We model secure geolocation as a three-party problem. First, there is the geolocation *user* or *victim*. The user hopes to accurately determine the location of the target using a geolocation algorithm that relies on measurements of network properties[1]. We assume that; (1) the user has access to a number of landmark machines distributed around the globe to make measurements of RTTs and network paths, and (2) the user trusts the results of measurements reported by landmarks. Second, there is the *adversary*, who owns the target's IP address. The adversary would like to mislead the user into believing that the target is at a *forged location* of the adversary's choosing, when in reality the target is actually located at the *true location*. The adversary is responsible for physically connecting the target IP address to the Internet, which allows them to insert additional machines or routers between the target and the Internet. The third party is the *Internet* itself. While the Internet is impartial to both adversary and user, it introduces additive noise as a result of queuing delays and circuitous routes. These properties introduce some inherent inaccuracy and unpredictability into the results of measurements on which geolocation algorithms rely. In general, an adversary's malicious tampering with network properties (such as adding delay), if done in small amounts, is difficult to distinguish from additive noise introduced by the Internet.

This work addresses two types of adversaries with differing capabilities. We assume in both cases that the adversary is fully aware of the geolocation algorithm and knows both the IP addresses and locations of all landmarks used in the algorithm. The first, *simple adversary* can tamper only with the RTT measurements taken by the landmarks. This can be done by selectively delaying packets from landmarks to make the RTT appear larger than it actually is. The simple adversary was chosen to resemble a home user running a program to selectively delay responses to measurements. The second, *sophisticated adversary*, controls several IP addresses and can use them to create fake routers and paths to the target. Further, this adversary may have a wide area network (WAN) with several gateway routers and can influence BGP routes to the target. The sophisticated adversary was chosen to model a cloud provider as the adversary. Many large online service providers already deploy WANs [11], making this attack model feasible with low additional cost to the provider.

We make two assumptions in this work. First, while aware of the geolocation algorithm being used, and the location and IP addresses of all landmarks, the adversary cannot compromise the landmarks or run code on them. Thus, the only way the adversary can compromise the integrity of network measurements is to modify the properties of traffic traveling on network links directly connected to a machine under its control.

The second assumption is that network measurements made by landmarks actually reach the target. Otherwise, an adversary could trivially attack the geolocation system by placing a proxy at the forged location that responds to all geolocation traffic and forwards all other traffic to the true location. To avoid this attack, the user can either combine the measurements with regular traffic or protect it using cryptography. For example, if the geolocation user is a Web content provider, Muir and Oorschot [18] have shown that even an anonymization network such as Tor [28] may be defeated using a Java applet embedded in a Web page. Users who want to geolocate a VM in a compute cloud may require the cloud provider to support tamper-proof VMs [10, 25] and embed a secret key in the VM for authenticating end-to-end network measurements. In this case, the adversary would need to place a copy of the VM in the forged location to respond to measurements. Given that the adversary is trying to avoid placing a VM in the forged location, it is not a practical attack for a malicious cloud provider.

## 4 Delay-based geolocation

Delay-based geolocation algorithms use measurements of end-to-end network delays to geolocate the target IP. To execute delay-based geolocation, the landmarks need to calibrate the relationship between geographic distance and network delay. This is done by having each landmark, $L_i$, ping all other landmarks. Since the landmarks have known geographic locations, $L_i$ can then derive a function mapping geographic distance, $g_{ij}$, to network delay, $d_{ij}$, observed to each other landmark $L_j$ where $i \neq j$ [12]. Each landmark performs this calibration and develops its own mapping of geographic distance to network delay. After calibrating its distance-to-delay function, it then pings the target IP. Using the distance-to-delay function, the landmark can then transform the observed delay to the target into a predicted distance to the target. All landmarks perform this computation to triangulate the location of the target.

Delay-based geolocation operates under the implicit assumption that network delay is well correlated with geographic distance. However, network delay is composed of queuing, processing, transmission and propagation delay [15]. Where only the propagation time of network traffic is related to distance traveled, and the other components vary depending on network load, thus adding noise to the measured delay. This assumption is also violated when network traffic does not take a direct ("as the crow flies") path between hosts. These indirect paths are referred to as "circuitous" routes [30].

There are many proposed methods for delay-based geolocation, including GeoPing [19], Statistical Geolocation [31], Learning-based Geolocation [9] and CBG [12]. These algorithms differ in how they express the distance-to-delay function and how they triangulate the position of the target. GeoPing is based on the observation that hosts that are geographically close to each other will have delay properties similar to the landmark nodes [19]. Statistical Geolocation develops a joint probability density function of distance to delay that is input into a force-directed algorithm used to geolocate the target [31]. In contrast, Learning-based Geolocation utilizes a Naïve Bayes framework to geolocate a target IP given a set of measurements [9]. CBG has the highest reported accuracy of the delay-based algorithms, with a mean error of 78-182 km [12]. The remainder of this section therefore focuses on CBG to model and evaluate how an adversary can influence delay-based geolocation techniques.

CBG [12] establishes the distance-delay function, described above, by having the landmarks ping each other to derive a set of points ($g_{ij}$,$d_{ij}$) mapping geographic distance to network delay. To mitigate the effects of congestion on network delays, multiple measurements are made, and the 2.5-percentile of network delays are used by the landmarks to calibrate their distance-to-delay mapping. Each landmark then computes a linear ("best line") function that is closest to, but below, the set of points. Distance between each landmark and the target IP is inferred using the "best line" function. This gives an implied circle around each landmark where the target IP may be located. The target IP is then predicted to be in the region of intersection of the circles of all the landmarks. Since the result of this process is a *feasible region* where the target may be located, CBG determines the centroid of the region and returns this value as the geolocation result. Gueye *et al.* observe a mean error of 182 km in the US and 78 km in Europe. They also find that the feasible region where the target IP may be located ranges from $10^4$ km$^2$ in Europe to $10^5$ km$^2$ in North America.

### 4.1 Attack on delay-based geolocation

Since delay-based geolocation techniques do not take network topology into account, the ability of a sophisticated adversary to manipulate network paths is of no additional value. Against a delay-based geolocation algorithm, the simple and sophisticated adversaries have equal power.

To mislead delay-based geolocation, the adversary can manipulate distance of the target computed by the landmarks by altering the delay observed by each landmark. The adversary knows the identities and locations of each landmark and can thus identify traffic from the landmarks and alter the delay as necessary. To make the target at the true location, $t$, appear to be at forged location, $\tau$, the adversary must alter the perceived delay, $d_{it}$, between each landmark, $L_i$ and $t$ to become the delay, $d_{i\tau}$, each landmark should perceive between $L_i$ and $\tau$. To do this, two problems must be solved. The adversary must first find the appropriate delay, $d_{i\tau}$, for each landmark and then change the perceived delay to the appropriate delay.




Figure 1: Landmarks (PlanetLab nodes) used in evaluation.


Figure 2: Forged locations ($\tau$) used in the evaluation.

If the adversary controls a machine at or near $\tau$, she may directly acquire the appropriate $d_{i\tau}$ for each landmark by pinging each of the landmarks from the forged location $\tau$. However, pings to all the landmarks from a machine not related to the geolocation algorithm may arouse suspicion. Also, it may not be the case that the adversary controls a machine at or near $\tau$.

Alternatively, with knowledge of the location of the landmarks, the adversary can compute the geographic distances $g_{it}$ and $g_{i\tau}$ between each landmark $L_i$ and the true location $t$ as well as the forged location $\tau$. This enables the adversary to determine the additional distance a probe from $L_i$ would travel ($\gamma_i = g_{i\tau} - g_{it}$) had it actually been directed to the forged location $\tau$. The next challenge is to map $\gamma_i$ into the appropriate amount of delay to add. To do this, the adversary may use 2/3 the speed of light in a vacuum ($c$) as a lower-bound approximation for the speed of traffic on the Internet [14]. Thus, the required delay to add to each ping from $L_i$ is:

$$\delta_i = \frac{2 \times \gamma_i}{2/3 \times c} \quad (1)$$

The additional distance the ping from $L_i$ would travel is multiplied by 2 because the delay measured by `ping` is the round-trip time as opposed to the end-to-end delay. This approximation is the lower bound on the delay that would be required for the ping to traverse the distance $2 \times \gamma_i$ because the speed of light propagation is the fastest data can travel between the two points.

Armed with this approximation of the appropriate $d_{i\tau}$ for each landmark, the adversary can now increase the delay of each probe from the landmarks. The perceived delay cannot be decreased since this would require the adversary to either increase the speed of the network path between $t$ and $L_i$, or slow down probes from $L_i$ during its calibration phase. Since the adversary cannot compromise the landmarks and does not control network paths that are not directly connected to one of her machines, she is not able to accomplish this. As a result, the adversary may only modify landmark delays that need to be increased (i.e., $d_{i\tau} > d_{it}$). For all other landmarks, she does not alter the delays. Thus, even with perfect knowledge of the delays $d_{i\tau}$, neither a simple nor sophisticated adversary will be able to execute an attack perfectly on delay-based geolocation techniques.

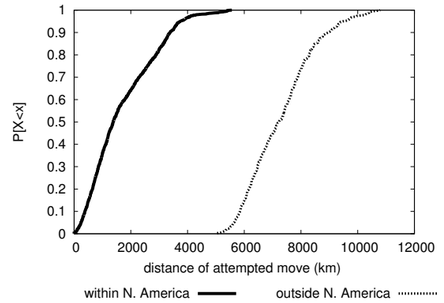



Figure 3: CDF of the distance the adversary tries to move the target.

### 4.2 Evaluation

We evaluate the effectiveness of our proposed attack against a simulator that runs the CBG algorithm proposed by Gueye *et al.* [12]. We collected measurement inputs for the algorithm using 50 PlanetLab nodes. Each node takes a turn being the target with the remaining 49 PlanetLab nodes being used as landmarks. Figure 1 shows the locations of the PlanetLab nodes. Each tar-

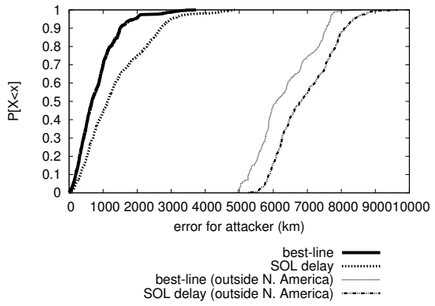



Figure 4: CDF of error distance for the adversary when attacking delay-based geolocation using speed of light (SOL) or best line delay.

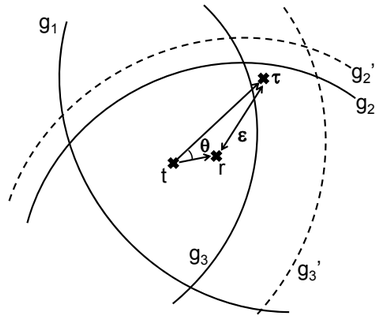



Figure 5: Attacking delay-based geolocation.

get is initially geolocated using observed network delays. The target is then moved to 50 forged locations using the delay-adding attack, shown in Figure 2. We select 40 of the forged locations based on the location of US universities and 10 based on the location of universities outside of North America. This results in a total of 2,500 attempted attacks on the CBG algorithm.

In the delay adding attack, the adversary cannot move a target that is not within the same region as the landmarks into that region. For example, if the target is located in Europe, moving it to a forged location in North America would require reducing delay to all landmarks, which is not possible. This implies that if a geolocation provider wants to prevent the adversary from moving the target into a specific region, it should place their landmarks in this desired region.

Figure 3 shows the CDF of the distances the adversary attempts to move the target. In North America, the target is moved less than 4,000 km most of the time moved moved less than 1,379 km 50% of the time. Outside of North America, the distance moved consistently exceeds 5,000 km.

We evaluate the delay-adding attack under two circumstances: (1) when the adversary knows exactly what delay to add (by giving the adversary access to the "best line" function used by the landmarks), and (2) when the adversary uses the speed of light (SOL) approximation for the additional delay.

#### 4.2.1 Attack effectiveness

Since the adversary is only able to increase, and not decrease, perceived delays, there are errors between the forged location, $\tau$, and the actual location, $r$, returned by the geolocation algorithm. To understand why these errors exist, consider Figure 5. The arcs labeled $g_1$, $g_2$, and $g_3$ are the circles drawn by 3 landmarks when geolocating the target. The region enclosed by the arcs is the feasible region, and the geolocation result is the centroid of that region. To move $t$ to $\tau$, the adversary should increase the radii of $g_2$ and $g_3$ and decrease the radius of $g_1$. However, as described earlier, delay can only be added, meaning that the adversary can only increase the radii of $g_2$ and $g_3$ to $g_2'$ and $g_3'$, respectively (shown by the dotted lines). Since the delay of $g_1$ cannot be decreased, this results in a larger feasible region with a centroid $r$ that does not quite reach $\tau$. We call the difference between the geolocation result ($r$) and forged location ($\tau$) the error distance ($\varepsilon$) for the adversary. The difference between the intended and actual direction of the move is the angle $\theta$.

We begin by evaluating the error distance, $\varepsilon$. Figure 4 shows the CDF of error for the adversary over the set of attempted attacks in our evaluation. Within North America, an adversary using the speed of light approximation has a median error of 1,143 km. When the adversary has access to the best line function,their error decreases to 671 km. As a reference, 671 km is approximately half the width of Texas. This indicates that when moving within North America, it is possible for an adversary with access to the best line function to be successful in trying to move the target into a specific state. We note that three of the targets used in our evaluation were located in Canada. Using the speed of light approximation these Canadian targets are able to appear in the US 65% of the time. Using the best line function, they are able to move into the US 89% of the time.

Outside of North America, the delay-adding attack has poor accuracy with a minimum error for the adversary of 4,947 km. As a reference, the distance from San Francisco to New York City is 4,135 km. Error of this magnitude is not practical for an adversary attempting to place the target in a specific country. For the remainder of this section, we focus on attacks where the adversary tries to move within North America because the error for the adversary is more reasonable.



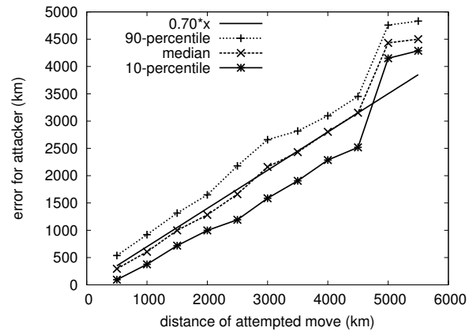



Figure 6: Error observed by the adversary depending on distance of their attempted move for the delay-adding attack.

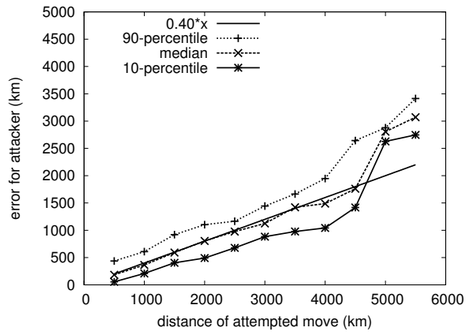



Figure 7: Error observed by the adversary depending on distance of their attempted move for the delay-adding attack when they have access to the best line function.

We next consider how the distance the adversary tries to move the target affects the observed error. Figure 6 shows error for the adversary depending on how far the adversary attempts to move the target when using the speed of light approximation. Figure 7 shows the same data for an adversary with access to the best line function. We note that the error observed by the adversary grows with the magnitude of the attempted move by the adversary. Specifically, for each 1 km the adversary tries to move the median error increases by 700 meters when she does not have access to the best line function. With access to the best line function, the median error per km decreases by 43% to 400 km. Thus, the attack we propose works best when the distance between $t$ and $\tau$ is relatively small and the error observed by the attacker grows linearly with the size of the move.

Given the relatively high errors observed by the adversary, we next verify whether the adversary moves in her chosen direction. Figure 8 shows the CDF of $\theta$, the difference between the direction the adversary tried to move and the direction the target was actually moved. While lacking high accuracy when executing the delay-adding attack, the adversary is able to move the target in the general direction of her choosing. The difference in direction is less than 45 degrees 74% of the time and less than 90 degrees 89% of the time. The attack where the adversary has access to the best line function performs better with a difference in direction of less than 45 degrees 91% of the time.

#### 4.2.2 Attack detectability

We next look at whether a geolocation provider can detect the delay-adding attack and thus determine that the geolocation result has been tampered with.

When CBG geolocates a target, it determines a feasible region where the target can be located [12]. The size of the feasible region can be interpreted as a measure of confidence in the geolocation result. A very large region size indicates that there is a large area where the target may be located, although the algorithm returns the centroid. As we saw in Figure 5, the adversary, able only to add delay, can only increase the radii of the arcs and thus only increase the region size. As a result, the delay-adding attack always increases the feasible region size and reduces confidence in the result of the geolocation algorithm. We consider the region size computed by CBG before and after our proposed attack to determine how effective region size may be for detecting an attack.

Figure 9 shows the region size for CBG when the delay-adding attack is executed in general, when the attack only attempts to move the landmark less than 1,000 km, and where the adversary has access to the best line function. We observe that the region size becomes orders of magnitude larger when the delay-adding attack is executed. The region size grows even larger when the adversary uses the best line function. An adversary that moves the target less than 1,000 km is able to execute the attack without having much impact on the region size distribution.

The region size grows in proportion to the amount of delay added. This explains why the adversary creates a larger region size when using the best line function, which adds more delay than the speed of light approxi-

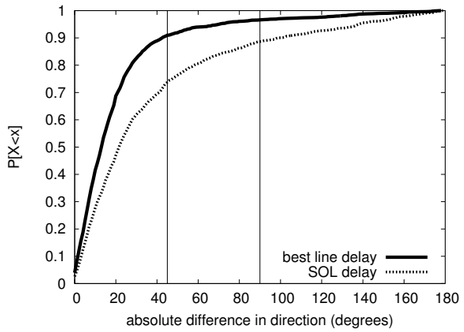



Figure 8: CDF of change in direction for the delay-adding attack.

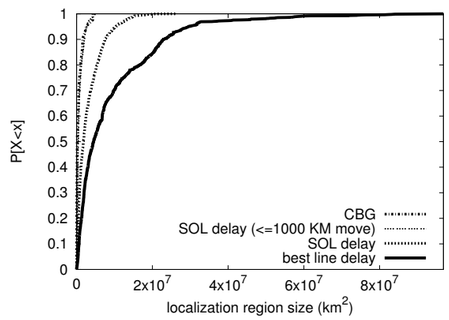



Figure 9: CDF of region size for CBG before and after the delay-adding attack.

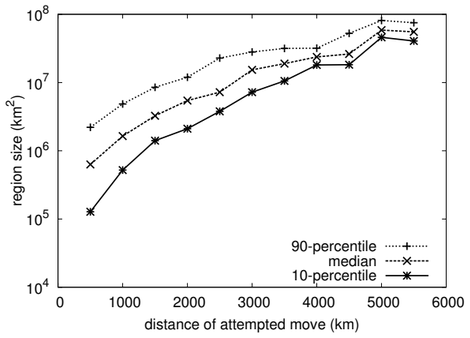



Figure 10: Region size depending on how far the adversary attempts to move the target using the best line function.

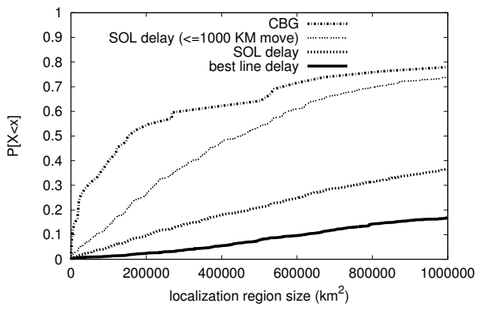



Figure 11: CDF of region size for CBG before and after delay-adding, limited to points less than 1,000,000 km$^2$.

mation. Figure 10 illustrates this case. As the adversary attempts to move the target further from its true location, the amount of delay that must be added increases. This in turn increases the region size returned by CBG. Thus, while there may be methods for adding delay that improve the adversary's accuracy, they will only increase the ability of the geolocation provider to detect the attack.

Given the increased region sizes observed when the delay-adding attack is executed, one defense would be to use a region size threshold to exclude geolocation results with insufficient confidence. Increased region sizes may be caused by an adversary adding delays, as we have observed or by fluctuations in the stochastic component of network delay. In either case, the geolocation algorithm observes a region that is too large for practical purposes. Suppose we discard all geolocation results with a region size greater than 1,000,000 km$^2$ (this is approximately the size of Texas and California combined). Figure 11 shows the CDF of region size below this threshold. The adversary using the speed-of-light approximation will be undetected only 36% of the time. However, if the adversary attempts to move less than 1,000 km she will remain undetected 74% of the time. An adversary with access to the best line for each of the landmarks is more easily detectable because of the larger region sizes that result from the larger injected delays. With a threshold of 1,000,000 km$^2$, the adversary using the best line function will have her results discarded 83% of the time. Thus, using a threshold on the region size is effective for detecting attacks on delay-based geolocation except when the attacker tries to move the target only a short distance.



# 5 Topology-aware geolocation

Delay-based geolocation relies on correlating measured delays with distances between landmarks. As we saw previously, these correlations or mappings are applied to landmark-to-target delays to create overlapping confidence regions; the overlap is the feasible region, and the estimated location of the target is its centroid. When inter-landmark delays and landmark-to-target delays are not similarly correlated with physical distances (e.g., due to circuitous end-to-end paths) the resulting delay-to-distance relationships to the target can deviate significantly from the pre-computed correlations.

Topology-aware geolocation addresses this problem by limiting the impact of circuitous end-to-end paths; specifically, it localizes all intermediate routers in addition to the target node, which results in a better estimate of delays. Starting from the landmarks, the geolocation algorithm iteratively estimates the location of all intermediate routers on the path between the landmark and the target. This is done solely based on single-hop link delays, which are usually significantly less circuitous than multi-hop end-to-end paths, enabling topology-aware geolocation to be more resilient to circuitous network paths than delay-based geolocation.

There are two previously proposed topology-aware geolocation methods, topology-based geolocation (TBG) [14] and Octant [30]. These methods differ in how they geolocate the intermediate routers. TBG uses delays measured between intermediate routers as inputs to a constrained optimization that solves for the location of the intermediate routers and target IP [14]. In contrast, Octant leverages a "geolocalization" framework similar to CBG [12], where the location of the intermediate routers and target are constrained to specific regions based on their delays from landmarks and other intermediate routers [30]. These delays are mapped into distances using a convex hull rather than a linear function, such as the best line in CBG to improve the mapping between distance and delay.

Octant leverages several optimizations that improve its performance over other geolocation algorithms. These include: taking into account both positive and negative constraints; accounting for fixed delays along network paths, and decreasing the weight of constraints based on latency measurements. Wong *et al.* find that their scheme outperforms CBG, with median accuracies of 35-40 km [30]. In addition, the feasible regions returned by Octant are much smaller than those returned by CBG. They also observe that their scheme is robust even given a small number of landmarks with performance leveling off after 15 landmarks.

When analyzing and evaluating attacks on topology-aware geolocation, we consider a generic geolocation framework. Intermediate routers are localized using constraints generated from latencies to adjacent routers. The target is localized to a feasibility region generated based on latencies from the last hop(s) before the target, and the centroid of the region is returned.

## 5.1 Delay-based attacks on topology-aware geolocation

Topology-aware geolocation systems localize all intermediate routers in addition to the target node. We begin by analyzing how a simple adversary, one without the ability to fabricate routers, could attack the geolocation system, and then move onto how a sophisticated adversary could apply additional capabilities to improve the attack. Since the simple adversary has no control over the probes outside her own network, any change made can only be reflected on the final links of the path towards the target.

Most networks are usually connected to the rest of the Internet via a small number of gateway routers. Any path connecting nodes outside the adversary's network to the target (which is inside the network) will go through one of these routers. Here, we start with a simple case where all routes towards the target converge on a single gateway router; we then consider the more general case of multiple gateway routers.

CLAIM: 1 *If the network paths from the landmarks to the target converge to a single common gateway router, increasing the end-to-end delays between the landmarks and the target can be detected and mitigated by topology-aware geolocation systems.*

To verify this claim, we first characterize the effect of delay-based attacks on topology-aware geolocation. Delay-based attacks selectively increase the delay of the probes from landmarks. The probe from landmark $L_i$ is delayed for an additional $\delta_i$ seconds. Given that all network paths to the target converge to a single common gateway router $h$, the end-to-end delay from each landmark, $L_i$, to the target can be written as:

$$d_{it} = d_{ih} + d_{ht} + \delta_i \quad (2)$$

The observed latency from the gateway to the target is $d_{it} - d_{ih}$, which is the sum of the real last-hop latency and the attack delay. However, since the delay-based attack relies on selectively varying the attack delays, $\delta_i$, based on the location of $L_i$, the observed last-hop latency between the gateway and the target will be inconsistent across measurements initiated from different landmarks.

The high-variance in the last-hop link delay can be used to detect delay-based attacks in topology-aware geolocation systems. The attack can be mitigated by taking

the minimum observed delay for each link. The resulting observed link delay from $h$ to the target is:

$$\hat{d}_{ht} = d_{ht} + \min_{L_i \in L} \delta_i \quad (3)$$

This significantly reduces the scope of delay-based attacks, requiring attack delays to be uniform across all measurement vantage points when there is only a single common gateway to the target.

In general, if there are multiple gateway routers on the border of the adversary's network, we can make the following weaker claim:

CLAIM: 2 *Increasing the delay between each gateway and the target can only be as effective against topology-based geolocation as increasing end-to-end delays against delay-based geolocation with a reduced set of landmarks.*

An adversary could attempt to modify delays between each gateway router, $h_j$, and the target, $t$. This assumes the adversary knows the approximate geolocation results for all gateway routers [2]. Where there is only a single gateway router with no additional attack delay, topology-based geolocation places the target within a circle centered at $h$ with coordinates $(\lambda_h, \phi_h)$:

$$\sqrt{(x-\lambda_h)^2 + (y-\phi_h)^2} = d_{ht} \quad (4)$$

Subjecting the latency measurement to an additional delay, $\delta$, changes the equation to the following:

$$\sqrt{(x-\lambda_h)^2 + (y-\phi_h)^2} = d_{ht} + \delta \quad (5)$$

Thus, for targets with a single gateway router, an adversary can only increase the localization region by introducing an additional delay without changing the location of the region's geometric center.

For targets with multiple gateway routers $H = h_0, h_1, ..., h_n$, targets are geolocated based on the delays between the gateways and $t$. An adversary can add additional delay, $\delta_j$, between each gateway, $h_j$, and $t$ based on the location of $h_j$. This is equivalent to the delay-adding attack, except the previously geolocated gateway routers are used in place of the real landmarks. Therefore, the previous evaluation results for the delay-adding attack on delay-based geolocation can be extended to topology-based geolocation for targets with multiple gateway routers.

## 5.2 Topology-based attacks

In topology-based geolocation, intermediate nodes are localized to confidence regions, and geographic constraints constructed from these intermediate nodes are expanded by their confidence regions to account for the accumulation of error. However, this does not result in a monotonic increase in the region size of intermediate nodes with each hop. The intersection of several expanded constraints for intermediate nodes along multiple network paths to the target can still result in intermediate nodes that are localized to small regions. A sophisticated adversary with control over a large administrative domain can exploit this property by fabricating nodes, links and latencies within its network to create constraint intersections at specific locations. This assumes that the adversary can detect probe traffic issued from geolocation systems in order to present a topologically different network without affecting normal traffic.

Externally visible nodes in an adversary's network consist of gateway routers $ER = \{er_0, er_1, ..., er_m\}$, internal routers $F = \{f_0, f_1, ..., f_n\}$ and end-points $T = \{\tau_0, \tau_1, ..., \tau_s\}$. Internal routers can be fictitious, and network links between internal routers can be arbitrarily manufactured. The adversary's network can be described as the graph $G = (V, E)$, where $V = F \cup ER \cup T$ represents routers, and $E = \{e_0, e_1, ..., e_k\}$ with weights $w(e_i)$ is the set of links connecting the routers with weights representing network delays.

All internal link latencies, including those between gateways, can be fabricated by the adversary. However, the delay between fictitious nodes must respect the speed-of-light constraint, which dictates that a packet can only travel a distance equal to the product of delay and the speed-of-light in fiber.

CLAIM: 3 *Topology-based attacks require the adversary to have more than one geographically distributed gateway router to its network.*

This claim follows from the analysis of delay-based attacks when all network paths to the target converge to a common gateway router. With only one gateway router to the network, changes to internal network nodes can affect only the final size of the localization region, not the region's geometric center.

CLAIM: 4 *An adversary with control over three or more geographically distributed gateway routers to its network can move the target to an arbitrary location.*

Unlike delay-based attacks that can only increase latencies from the landmarks to the target, topology-based attacks can assign arbitrary latencies from the ingress points to the target. From geometric triangulation, this enables topology-based attacks to, theoretically, triangulate the location of the target to any point on the globe given three or more ingress points.

In practice, there are challenges that limit the adversary from achieving perfect accuracy with this attack. Specifically, the attack requires the adversary to know the estimated location of the gateway routers and to have an accurate model of the delay-to-distance function used by the geolocation system. Such information can be reverse-engineered by a determined adversary by analyzing the geolocation results of other targets in the adversary's network.

Although a resourceful adversary's topology-based attack can substantially affect geolocation results, it can also introduce additional *circuitousness* to all network paths to the target that creates a detectable signature. Circuitousness refers to the ratio of actual distance traveled along a network path to the direct distance between the two end points of a path. Circuitousness can be observed by plotting the location of intermediate nodes as they are located by the topology-aware geolocation system.

### 5.2.1 Naming attack extension

State-of-the-art, topology-based geolocation systems [14, 30] leverage the structured way in which most routers are named to extract more precise information about router location. A collection of common naming patterns is available through the *undns* tool [27], which can extract approximate city locations from the domain names of routers.

When geolocation relies on `undns`, an adversary can effectively change the observed location of the target even with only a single gateway router to its network. This naming attack requires the adversary is capable of crafting a domain name that can deceive the *undns* tool, poisoning the *undns* database with erroneous mappings or responding to traceroutes with a spoofed IP address. The adversary only needs to use the naming attack to place any last hops before the target at its desired geographic location. The target will then be localized to the same location as this last hop in the absence of sufficient constraints.

Naming attacks exhibit the same increased circuitousness as standard topology-based attacks. Extensive poisoning of the *undns* database could allow an attacker to change the location of other routers along the network paths to reduce path circuitousness.

## 5.3 Evaluation

We evaluate the topology-based (hop-adding) attack and *undns* naming extension using a simulator of topology-aware geolocation. To perform the evaluation, we developed the fictitious network illustrated in Figure 12. The network includes 4 gateway routers ($ER$), represented by PlanetLab nodes in Victoria, BC; Riverside, CA; Ithaca, NY, and Gainesville, FL. The network also includes 11 forged locations ($T$) and 14 non-existent internal routers ($F$). Three of the non-existent routers are geographically distributed around the US, while the other 11 are placed close to the forged locations to improve the effectiveness of the attack, especially when the adversary can manipulate *undns* entries. Routers in the fictitious network are connected using basic heuristics. For example, each of the 11 internal routers near the forged locations is connected to the 3 routers nearest them to aid in triangulation. We show that even using this simple network design, an adversary executing the hop-adding attack and *undns* extension can be successful.

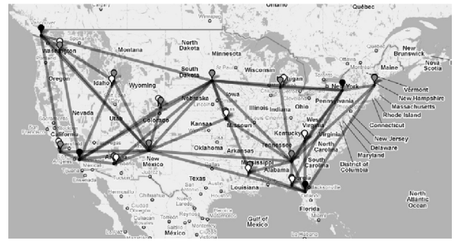



Figure 12: The adversary's network used for evaluating the topology-based attack.

To evaluate the attack, we use the same set of 50 PlanetLab nodes used in evaluating the delay-adding attack (Figure 1), with an additional 30 European PlanetLab nodes that act only as targets attempting to move into North America. We move the targets to the 11 forged locations in the fictitious network. These locations, a subset of the 40 US locations used in evaluating the delay-adding attack, were chosen to be geographically distributed around the US. Each of the 80 PlanetLab nodes takes a turn being the target with the remaining US PlanetLab nodes used as landmarks. Each target is moved to each of the 11 forged locations in turn, for a total of 880 attacks.

When executing the attack, the traceroute from each landmark is directed to its nearest gateway router. The first part of the traceroute is dictated by the network path between the landmark and its nearest gateway router (represented by a PlanetLab node). The second part is artificially generated to be the shortest path between the gateway router and the forged location. The latency of the second part is lower bounded by the speed-of-light delay between the gateway router and the target's true location. When the speed-of-light latency between the gateway router and the target is greater than the latency on the shortest path from the gateway to the forged location, the additional delay is divided across links in the shortest path.

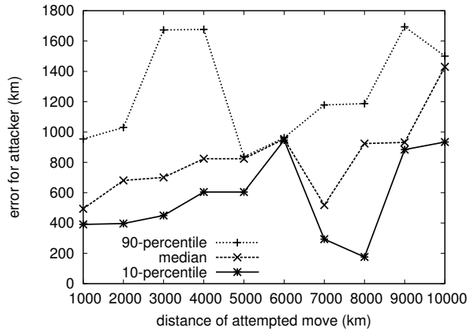



Figure 14: Error observed by the adversary depending on how far they attempt to move the target using the topology-based attack.

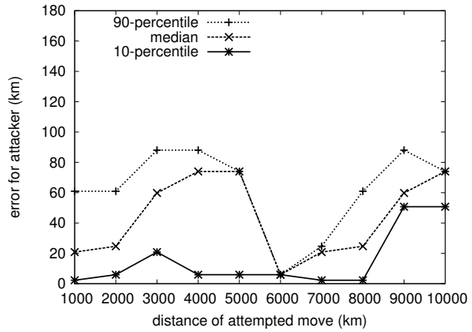



Figure 15: Error observed by the adversary depending on how far they attempt to move the target using the *undns* attack.

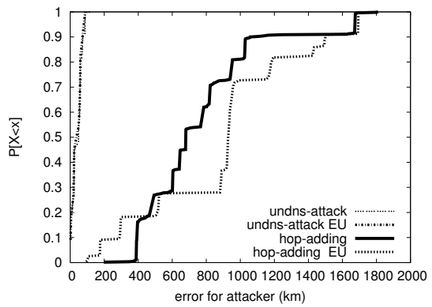



Figure 13: CDF of error distance for the attacker when executing the topology-based and *undns* attacks.

### 5.3.1 Attack effectiveness

We begin by examining how accurate the adversary can be when attempting to move the target to a specific forged location. Figure 13 shows the error for the adversary when executing the topology-based attack and *undns* extension. Without the *undns* extension, the adversary is able to place a North American target within 680 km of the false location 50% of the time. This is similar to the delay-adding attack in which the adversary has access to the best line function. When moving a target from Europe to North America, the adversary's median error increases by 50% to 929 km. Despite this increase, we observe that the adversary succeeds in each attempt to move a European target into the US. In addition to the overall decrease in accuracy for the adversary, we note that there are some instances where the target in Europe misleads the algorithm with higher accuracy. This is caused by the adversary using the speed-of-light approximation for latencies within their network. Since the speed-of-light is the lower bound on network delay, when additional delay is added to the links to account for the time it would take a probe to reach the target in Europe, the delay approaches the larger delay expected by the landmarks' distance-to-delay mapping. The *undns* extension increases the adversary's accuracy by 93%, with the adversary locating herself within 50 km of the forged location 50% of the time. These results are consistent whether the true location of the target is in North America or Europe.

When analyzing the delay-adding attack, we observed a linear relationship between the distance the adversary attempts to move the target and the error she observes. Figures 14 and 15 show the 10th percentile, median and 90th percentile error for the attacker depending on how far the forged location is from the target for the topology-based attack and *undns* extension, respectively. The observed errors were quite erratic which is a result of the many other factors that affect the accuracy of geolocation beyond the distance of the attempted move. In general, error for the adversary increases slowly as the adversary tries to move the target longer distances. This enables an adversary executing the topology-based attack to move the target longer distances. Error for the adversary using the *undns* extension remains fairly constant regardless of how far they attempt to move the target. In the case of the *undns* attack, the median accuracy fluctuates by less than 60 km whether the adversary moves 500 km or 4,000 km. The slow growth of adversary error stems from the engineered delays in the fictitious network. These delays cause nodes along the paths (including the end point) to



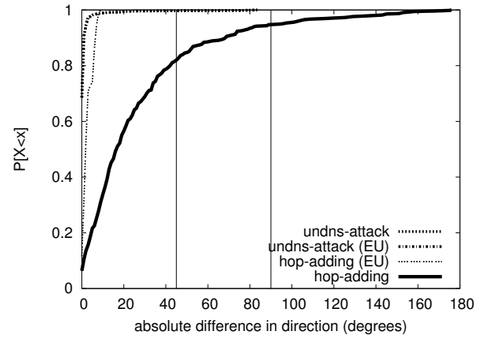



Figure 16: CDF of change in direction for the topology-based and *undns* extension.

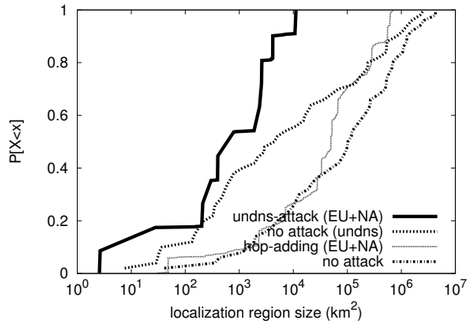



Figure 17: CDF of region size before and after the topology-based attack and *undns* extension.

be geolocated to a similar location regardless of where the target location was originally located.

We next confirm that the adversary is able to move in her chosen direction. Figure 16 shows the difference between the direction the adversary tried to move the target and the direction the target was actually moved ($\theta$ in the delay-adding attack). For the general topology-based attack, the adversary is within 36 degrees of her intended direction 75% of the time and within 69 degrees 90% of the time. This improves with the *undns* extension where the adversary is within 3 degrees of their intended direction 95% of the time. When the target attempts to move from Europe to North America, they always move very close to their chosen direction. The adversary always is within 10 degrees of her chosen direction. The smaller change in direction for European nodes stems from the longer distance between the target and the forged location. This causes a smaller change in direction to be observed for similar error values compared to a target that is closer to the forged location.

### 5.3.2 Attack detectability

We have observed that an adversary executing the topology-based attack and the *undns* extension to the attack can accurately relocate the geolocation target. We next consider whether the victim would be able to detect these attacks and reduce their impacts on geolocation results.

Figure 17 shows the region sizes for topology-aware geolocation and *undns* geolocation before and after the attacks are executed (for both North America and European targets). Unlike the delay-adding attack, the adversary that adds hops to the traceroutes of the victim has region sizes similar to the original algorithms and, in some cases, even smaller region sizes. For topology-aware geolocation, we observe median region sizes of 102,273 km$^2$ before and 50,441 km$^2$ after the attack. For the *undns* extension, we observe median region sizes of 4,448 km$^2$ before and 790 km$^2$ after the attack. These results indicate that region size is a poor metric for ruling out attacks that add hops to the end of traceroute paths.

Another metric that may be used to rule out geolocation results that have been modified by an adversary is path *circuitousness*. We define circuitousness of a traceroute path between landmark, $L_i$, and the target as follows, where $r = (\lambda_r, \phi_r)$ is the location returned by the geolocation algorithm, and $h_j = (\lambda_j, \phi_j)$ is the location of intermediate hop $j$ as computed by the geolocation algorithm:

$$C = \frac{d_{ih_0} + \Sigma_{j=1}^{n} d_{h_{j-1}h_j} + d_{h_n r}}{d_{ir}} \qquad (6)$$

Figure 18 shows the distribution of circuitousness for paths between each landmark and the target for topology-aware geolocation before and after the topology-based attack is executed[3]. We observe that when the topology-based attack is executed the circuitousness per landmark increases. One criterion a geolocation algorithm can use for discarding results from the topology-based attack would be to discard results from landmarks where the circuitousness is abnormally high. If a geolocation framework that assigns weights to constraints, such as Octant, is used, constraints from landmarks with high circuitousness could be given a lower weight to limit the adversary's effectiveness. We note that a clever adversary could design her network to use more direct paths, making it more difficult to detect the attack by observing circuitousness.

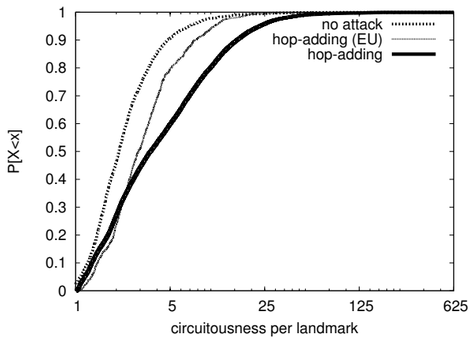



Figure 18: CDF of circuitousness for each landmark before and after the topology-based attack.

## 6 Related work

While there have been many related works on developing and evaluating geolocation algorithms (e.g., [12, 14, 26, 30]), there has been limited study of IP geolocation given a non-benign target [5, 18].

Castelluccia *et al.* consider the application of CBG [12] to the problem of geolocating hidden servers hosting illegal content within a botnet [5]. The technique used to hide these servers is referred to as "fast-flux", where a constantly changing set of machines infected by a botnet is used to proxy HTTP messages for a hidden server. Geolocating these servers is important to enable the appropriate authorities to take action against them. Castelluccia *et al.* leverage the fact that the hidden server is behind a layer of proxies to factor out the portion of the observed RTT caused by the proxy layer. They use HTTP connections to measure RTTs (because the hidden servers are unlikely to respond to `ping` ) and factor out additional delay caused by the layer of proxies to geolocate hidden servers with a median error of 100 km using PlanetLab nodes as ground truth hidden servers.

Muir and Oorschot survey a variety of geolocation techniques and their applicability in the presence of an adversarial target [18]. Their work is similar to but distinct from ours. Specifically, they emphasize geolocation techniques that leverage secondary sources of information, such as `whois` registries based on domain, IP and AS; DNS LOC [8]; application data from HTTP headers, and data inferred from routing information. They consider delay-based geolocation but do not specify or evaluate any attacks on measurement-based geolocation. Muir and Oorschot discuss the limitations of IP geolocation when an adversary attempts to conceal her IP address through the use of an anonymization proxy and examine how a Web page embedding a Java applet can discover a client's true identity using Java's socket class to connect back to the server. They demonstrate this strategy for identifying clients using the Tor [28] anonymization network.

These previous works begin to consider the performance of geolocation algorithms when the target of geolocation may have incentive to be adversarial. However, they generally focus on the issue of geolocating hosts that attempt to deceive geolocation using proxies. In contrast, we develop and evaluate attacks on two classes of measurement-based geolocation techniques by manipulating the network properties on which the techniques rely.

We observe that the problem of geolocating an adversarial target is similar to the problem of secure positioning [4] in the domain of wireless networks. Unlike wireless signals, network delay is subject to additive noise as a result of congestion and queuing along the network path as well as circuitous routes. Multiple hops along network paths on the Internet and the existence of large organizational WANs also enable new adversarial models in the domain of IP geolocation.

## 7 Conclusions

Many applications of geolocation benefit from security guarantees when confronted with an adversarial target. These include popular applications, such as limiting media distribution to a specific region, fraud detection, and newer applications, such as ensuring regional regulatory compliance when using an infrastructure as a service provider. This paper considered two models of an adversary trying to mislead measurement-based geolocation techniques that leverage end-to-end delays and topology information. To this end, we developed and evaluated two attacks against delay-based and topology-aware geolocation.

To avoid detection, adversaries can leverage inherent variability in network delay and circuitousness of network paths on the Internet to hide their tampering. Since these properties are measured and used by various geolocation techniques, they serve as good attack vectors by which the adversary can influence the geolocation result.

Our most surprising finding is that the more advanced and accurate topology-aware geolocation techniques are more susceptible to covert tampering than the simpler delay-based techniques. For geolocation algorithms that leverage delay, we observed how a simple adversary that only adds delay to probes could alter the results of geolocation. However, this adversary has limited precision when attempting to forge a specific location. We also observed a clear trade-off between the amount of delay an adversary added and her detectability, using the region size returned by CBG [12] as a metric for discarding anomalous results.

Compared to delay-based geolocation, topology-aware geolocation fares no better against a simple adversary and worse against a sophisticated one. Topology-aware geolocation uses more information sources, such as traceroute and *undns* , to achieve higher accuracy than delay-based geolocation. Unfortunately, this advantage becomes a weakness against an adversary able to corrupt these sources. A sophisticated adversary that can leverage multiple network entry points (e.g., an infrastructure as a service provider) can cause the geolocation system to return a result as accurate as the best case simple adversary without increasing the resultant region size. When *undns* entries are corrupted, the adversary is able to forge locations with high accuracy without increasing the region sizes – in some cases, even decreasing them.

Our work reveals limitations of current measurement-based geolocation techniques given an adversarial target. To provide secure geolocation, these algorithms must account for the presence of untrustworthy measurements. This may be in the form of heuristics to discount measurements deemed untrustworthy or through the use of secure measurement protocols. We intend to explore these directions in future work.

## Acknowledgements

The authors would like to thank the anonymous reviewers and our shepherd, Steven Gribble, for their feedback, which has helped to improve this paper. This work was supported by the Natural Sciences and Engineering Research Council (NSERC) ISSNet and NSERC-CGS funding.

## References

[1] Amazon EC2, 2010. `http://aws.amazon.com/ec2/`.

[2] ANDERSON, M., BANSAL, A., DOCTOR, B., HADJIYIANNIC, G., HERRINGSHAW, C., KARPLUS, E., AND MUNIZ, D. Method and apparatus for estimating a geographic location of a networked entity, June 2004. US Patent number: 6684250.

[3] American Registry for Internet numbers (ARIN), 2010. `http://www.arin.net`.

[4] CAPKUN, S., AND HUBAUX, J. Secure positioning of wireless devices with application to sensor networks. In *Proceedings of IEEE INFOCOM Conference* (March 2005).

[5] CASTELLUCCIA, C., KAAFAR, M., MANILS, P., AND PERITO, D. Geolocalization of proxied services and its application to fast-flux hidden servers. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference* (November 2009).

[6] CBC. USA Patriot Act comes under fire in B.C. report, October 2004. `http://www.cbc.ca/canada/story/2004/10/29/patriotact_bc041029.html`.

[7] CROVELLA, M., AND KRISHNAMURTHY, B. *Internet Measurement: Infrastructure, Traffic and Applications*. John Wiley & sons, 2006.

[8] DAVIS, C., VIXIE, P., GOODWIN, T., AND DICKINSON, I. A means for expressing location information in the domain name system. RFC 1876, IETF, Jan. 1996.

[9] ERIKSSON, B., BARFORD, P., SOMMERS, J., AND NOWAK, R. A learning-based approach for IP geolocation. In *Proceedings of the Passive and Active Measurement Workshop* (April 2010).

[10] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (October 2003).

[11] GILL, P., ARLITT, M., LI, Z., AND MAHANTI, A. The flattening Internet topology: Natural evolution, unsightly barnacles or contrived collapse? In *Proceedings of the Passive and Active Measurement Workshop* (April 2008).

[12] GUEYE, B., ZIVIANI, A., CROVELLA, M., AND FDIDA, S. Constraint-based geolocation of Internet hosts. *IEEE/ACM Transactions on Networking 14*, 6 (December 2006).

[13] Hulu - watch your favorites. anytime. for free., 2010. `http://www.hulu.com/`.

[14] KATZ-BASSET, E., JOHN, J., KRISHNAMURTHY, A., WETHERALL, D., ANDERSON, T., AND CHAWATHE, Y. Towards IP geolocation using delay and topology mesurements. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference* (October 2006).

[15] KUROSE, J., AND ROSS, K. *Computer Networking: A top-down approach featuring the Internet*. Addison-Wesley, 2005.

[16] Maxmind - geolocation and online fraud prevention, 2010. `http://www.maxmind.com`.

[17] M.CASADO, AND FREEDMAN, M. Peering through the shroud: The effect of edge opacity on IP-based client identification. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)* (Cambridge, MA, April 2007).

[18] MUIR, J., AND VAN OORSCHOT, P. Internet geolocation: Evasion and counterevasion. *ACM Computing Surveys 42*, 1 (December 2009).

[19] PADMANABHAN, V., AND SUBRAMANIAN, L. An investigation of geographic mapping techniques for Internet hosts. In *Proceedings of ACM SIGCOMM* (August 2001).

[20] Pandora Internet radio, 2010. `http://www.pandora.com`.

[21] Planetlab, 2010. `http://www.planet-lab.org`.

[22] Quova – IP geolocation experts, 2010. `http://www.quova.com`.

[23] Reseaux IP Europeens (RIPE), 2010. `http://www.ripe.net`.

[24] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off my cloud! exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)* (November 2009).

[25] SANTOS, N., GUMMADI, K. P., AND RODRIGUES, R. Towards trusted cloud computing. In *Proceedings of the 1st Workshop in Hot Topics in Cloud Computing (HotCloud)* (June 2009).

[26] SIWPERSAD, S., GUEYE, B., AND UHLIG, S. Assessing the geographic resolution of exhaustive tabulation. In *Proceedings of the Passive and Active Measurement Workshop* (April 2008).

[27] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP topologies with Rocketfuel. In *Proceedings of ACM SIGCOMM* (August 2002).

[28] THE TOR PROJECT. Tor: Overview, 2010. `http://www.torproject.org/overview.html.en`.

[29] TRANCREDI, P., AND MCCLUNG, K. Use case: Restrict access to online bettors, August 2009. http://www.quova.com/Uses/UseCaseDetail/09-08-31/Restrict_Access_to_Online_Bettors.aspx.

[30] WONG, B., STOYANOV, I., AND SIRER, E. G. Octant: A comprehensive framework for the geolocalization of Internet hosts. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)* (Cambridge, MA, April 2007).

[31] YOUNG, I., MARK, B., AND RICHARDS, D. Statistical geolocation of Internet hosts. In *Proceedings of the 18th International Conference on Computer Communications and Networks* (August 2009).

## Notes

[1] In reality, the consumer of geolocation information will likely contract out geolocation services from a third party geolocation provider that will maintain landmarks. Given the common goals of these two entities we model them as a single party.

[2] The adversary can assume that the gateway routers are geolocated to their true locations.

[3] We make similar observations for the *undns* attack extension.



# Idle Port Scanning and Non-interference Analysis of Network Protocol Stacks Using Model Checking

Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall
*University of New Mexico,*
*Dept. of Computer Science*
{*royaen, joumon, kapur, crandall*}*@cs.unm.edu*

## Abstract

Idle port scanning uses side-channel attacks to bounce scans off of a "zombie" host to stealthily scan a victim IP address or infer IP-based trust relationships between the zombie and victim. We present results from building a transition system model of a network protocol stack for an attacker, victim, and zombie, and testing this model for non-interference properties using model checking. Two new methods of idle scans resulted from our modeling effort, based on TCP RST rate limiting and SYN caches, respectively. Through experimental verification of these attacks, we show that it is possible to scan victims which the attacker is not able to route packets to, meaning that protected networks or ports closed by firewall rules can be scanned. This is not possible with the one currently known method of idle scan in the literature that is based on non-random IPIDs.

For the future design of network protocols, a notion of trusted *vs.* untrusted networks and hosts (based on existing IP-based trust relationships) will enable shared, limited resources to be divided. For a model complex enough to capture the details of each attack and where a distinction between trusted and untrusted hosts can be made, we modeled RST rate limitations and a split SYN cache structure. Non-interference for these two resources was verified with symbolic model checking and bounded model checking to depth 1000, respectively. Because each transition is roughly a packet, this demonstrates that the two respective idle scans are ameliorated by separating these resources.

## 1 Introduction

Network reconnaissance is the important first step of virtually all network attacks. By scanning the network, the attacker is able to gain valuable information about the hosts that exist and the services they offer, infer IP-based trust relationships between hosts that are enforced by firewall rules and router tables, and collect other information that they can use in the next stage of attack. In this paper, we show that model checking can be a useful framework for predicting and mitigating attacker capabilities. In idle scans, an attacker scans a victim without sending packets to that victim using its own return IP address. The model of idle scans that we describe in this paper led to the discovery of two new forms of idle scan[1]. One of these, based on SYN cache structures that are common to all modern network stacks, gives an attacker capabilities beyond the one currently known form of idle scan in the literature. We demonstrate that it is possible to infer the liveness of hosts and some information about what operating system they are running on a subnetwork without the ability to route packets to that network. We also demonstrate that it is possible to port scan a network on a port that the firewall protecting the network blocks. This means that if, *e.g.*, a particular port is blocked by a firewall for an entire subnetwork, an attacker can scan the hosts on that subnetwork on that port from outside the firewall using idle scans. Finally, we demonstrate that if a distinction between trusted and untrusted hosts were made explicit in the lower layers of the network protocol stack, then separate RST rate limitations and a split SYN cache structure eliminates these attacks in our model of network stacks, which is complex enough to model all of the details of each attack.

The two new forms of idle scan that have resulted from the model checking effort presented in this paper are based on RST rate limiting and SYN caches, respectively. These were discovered during the process of building the model and manifest as counterexamples to a non-interference property that are produced by the model checker. In the RST rate limiting counterexample, the zombie in this case is a FreeBSD machine that limits the number of RST packets that it will send in a given time period. The attacker can infer the port status of the victim by testing the rate at which the zombie will reply with RST packets, the details of this are in Section 4.

**The SYN cache counterexample is different from the existing IPID-based idle scan, which is described in Section 2, in that the attacker never sends packets to the victim, not even forged packets.** Instead the attacker forges SYN packets from the victim to the zombie, and the zombie sends a SYN/ACK to the victim and places these SYN packets in its SYN cache (a data structure for holding half-open TCP connections for which a SYN/ACK has been sent but an ACK response has yet to arrive). Because RSTs and ICMP errors from the victim will cause this SYN cache entry to be removed, the attacker can effectively perform a SYN/ACK scan of the victim without needing the ability to route packets to the victim. The attacker does this by testing the state of the SYN cache by sending SYNs with its own return IP address and viewing the SYN/ACK responses. The replies of the victim probes can be inferred from the attacker's ability to get SYN cache entries for its own SYNs. This makes possible testing for the liveness of IP addresses on protected networks with a rudimentary form of OS detection, and even port scanning on certain types of hosts on a port that is entirely blocked by a firewall. More details are given in Section 4.

Like virtually all side-channel attacks, idle scans are associated with shared, limited resources. Because these resources generally cannot be made unlimited, we recommend in light of our results that trust relationships between hosts be made explicit to those hosts all the way down to the IP layer. Currently the only distinction at the TCP and IP layers is subnetworks, which do not necessarily correspond to the IP-based trust relationships between hosts that are enforced by firewall rules and routing tables. Trusted hosts can be hosts protected by the same firewall or that have special trust relationships in the packets they can route to each other. By making a distinction between trusted and untrusted hosts non-interference can be achieved by statically dividing shared resources, effectively eliminating idle scans. We verify non-interference for our model with separate RST rate limitations using symbolic model checking. Then we demonstrate that our split SYN cache structure using bounded model checking to a depth of 1000 transitions has no violations of non-interference. This means that no practical attack for this counterexamples exists within the constraints of our model.

This paper is organized as follows. Section 2 gives more background and related works. Our model is described in Section 3, followed by a description of the counterexamples discovered during the process of building the model and some experimental results from their implementation in Section 4. We demonstrate that non-interference is achievable by distinguishing between trusted and untrusted hosts in Section 4.3. This is followed by discussion and future work in Section 5.

## 2 Background and related work

Because network reconnaissance is an important first step in most network attacks, a fair amount of previous work has focused on detecting port scans. Staniford *et al.* [32] use simulated annealing to detect stealthy scans. Leckie and Kotagiri [18] present a probabilistic approach for detecting port scans, and Muelder *et al.* [23] propose a visualization approach. The scan behavior of Internet worms has been studied [29, 36, 12, 35], as has the scan detection problem at the backbone level [31, 30] and measurements of port scans and their side effects at Internet telescopes [24]. Jung *et al.* [14] describe an approach based on sequential hypothesis testing. Gates [9, 10] and Kang *et al.* [15] consider the problem of stealth port scans based on using many distributed hosts (*e.g.*, a botnet) to perform the scan. To our knowledge, ours is the first study to model idle scans, which are a distinct stealth technique that, in addition to being used for stealth, can also can be used for inferring IP-based trust relationships. Passively identifying hosts that have no routable IP address and are hidden by network address translation [2, 17] is a related problem to idle scans, but assumes a very different threat model where some amount of traffic can be viewed passively by the attacker.

Idle scans were introduced by Antirez [1] in a 1998 posting to the bugtraq mailing list. The one currently known form of idle scan, based on non-random, sequential IPIDs of older network stacks, was also described in this posting and is described in more detail by Lyon [20]. An IPID is a unique identifier for each IP packet, used primarily for IP fragmentation. In early implementations of the IP protocol, the IPID was chosen sequentially by simply incrementing the IPID value for each packet. Antirez showed that this made it possible to perform an idle scan of a victim by using a third host, the zombie, which the attacker need not have control of, in a form of side-channel attack.

In this form of idle scan, the attacker queries the zombie for IP packet responses and observes the sequence of IPIDs in the zombie's responses. The attacker then sends one or more SYN packets to the victim on the target port to be scanned with the return IP address of the zombie and return port of a closed port on the zombie. If the victim replies to the SYN with a SYN/ACK, meaning the victim's port is open, then the zombie will reply to the victim with a TCP reset (RST) and the attacker will observe a discontinuity in the sequence of IPIDs that it receives from the zombie. If the victim port is closed, the SYN is dropped or replied to by the victim with a RST, which the zombie simply drops and no discontinuity is observed by the attacker. Thus, the attacker is able to infer the port status of the victim without revealing their return IP address to the victim. Furthermore, the attacker is able to infer trust relationships between the victim and zombie. For example, the attacker might infer that the victim only accepts connections from a particular trusted subnetwork by using a zombie on that subnetwork. This is the one known form of idle scan in the current literature. Modern network stacks randomize the IPID for security reasons not related to idle scans, so the zombie must be an older system for this known type of idle scan to work.

IPID-based idle scans have been implemented in nmap [20] using an algorithm that accounts for interference from other hosts and packet loss. Lyon [20] is a good resource for how this type of idle scan works in detail and the different uses of idle scans. FTP bounce scans [20] are currently the only known way to port scan a victim host or network without routing forged packets to that host or network from the attacker. These scans use a feature that has been largely discontinued in FTP server implementations because of the many potential abuses of it. FTP bounce scans require that the attacker be able to log into an FTP session on the zombie, and operate at layer 7 (the application layer) of the OSI network model, whereas the SYN cache idle scan that we describe operates at layers 3 and 4 (TCP/IP) and requires only that the attacker be able to route SYN packets to an open port on the zombie.

Non-interference [11] is a widely used concept of information flow security that has seen wide application for proving security properties of programs. The works that are most related to ours in this space are those that treat non-interference as two or more separate scenarios that must produce the same result from the attacker's view for non-interference to be demonstrated, *e.g.*, TightLip [38] or the work of McCamant and Ernst [21, 22]. We apply non-interference to network stacks in this paper. Non-interference proved to be a very fruitful model of information flow in this context, but for future work that might consider packet loss, packet delay, and other such factors, alternatives such as non-deducibility [33] may be necessary. For the modeling effort presented in this paper, which is based on an abstracted model of real networks that does not include packet loss and delay, non-interference proved to be a very useful property because it can be specified with Linear Temporal Logic (LTL). Treating the problem as a covert channel problem and studying object storage [16] and timing channels [37] is an attractive approach, but covert channel models assume collusion of the sender and receiver of information and do not capture in their models the sequences of events necessary to describe idle scans in a natural way.

The model checker that we chose for our study is the Symbolic Analysis Laboratory (SAL) [3]. SAL provides a SAT-based bounded model checker that allows for counterexamples to be easily interpreted as a trace through the states of the model, or, in our case specifically, a sequence of packets. SAL also provides a BDD-based symbolic model checker. Model checking has been applied to many properties of network protocols and their implementations where specific bugs lead to security vulnerabilities or availability issues, (*e.g.*, [8, 25, 13]). We have particularly patterned our analysis following Rushby's tutorials for modeling the Needham-Schroeder protocol [26] to identify Lowe's bug and the fault-tolerant algorithm for maintaining interactive consistency (Byzantine agreement) [27] as the transition systems for these problems seem similar to the ones for modeling port scanning and side-channel attacks in a protocol stack. Our results demonstrate that model checking is also useful for studying information flow on networks, particularly in this paper within the context of idle port scans.

## 3 Formalizing non-interference analysis of idle scans

In this section, we first describe the basics of our network stack model, then we describe more details and its implementation in SAL, and finally we list simplifying assumptions of the model.

### 3.1 Modeling the network stack

A host is viewed to be at the end of the network, *i.e.*, an end host. Hosts have internal state, such as a SYN cache, RST rate limit variables, and receive buffers. Hosts also have ports, which can be open, closed or filtered and their status does not change. An open port is one that the host will accept an incoming TCP connection on. For UDP, open ports simply drop packets and closed ports send ICMP errors. Filtered ports behave as would a typical host, but for the results presented in this paper all ports are either open or closed and never filtered. Hosts reply to packets based on rules that model a typical Linux or FreeBSD network stack. Our model is based on the IP protocol and includes TCP (but only up to the point of half-open connections), ICMP, and UDP.

The SYN cache on a host is a cache for pending SYN packets for which a SYN/ACK has been sent and the host is waiting for an ACK to complete the TCP three-way handshake. The SYN cache drops duplicate SYNs for the same IP address and port pairs. In our model packets are only removed from the SYN cache when a TCP RST is received from the source IP address and port of the original SYN packet (because we only model half-open TCP/IP connections, so there is no ACK for the third part of a three-way TCP handshake). When the SYN cache is full, the host replies with a SYN cookie and drops the SYN. A SYN cookie is a method for sending an initial

sequence number in the SYN/ACK that, when ACKed by the remote host, contains enough information to complete the connection so that no state about the half-open connection needs to be kept in memory [4]. TCP RST rate limiting, where the number of resets sent by a host is limited, is based on the FreeBSD implementation where separate rate limits are maintained for open ports and closed ports.

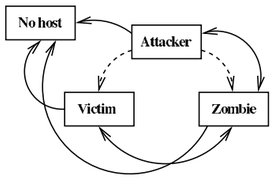



Figure 1: **Basic definition of an idle scan.**

Figure 1 shows the basic definition of an idle scan that we use for our model. There are four IP addresses, three for the attacker, zombie, and victim hosts, and one where there is no host so all packets are dropped. A solid arrow denotes that the source host can send packets to the destination using its own return IP address. A dashed arrow indicates that the source can send a packet to the destination using any return IP address other than its own. The salient feature of this definition of an idle scan is that the attacker cannot send packets to the victim using its own return IP address. This entails that the victim never sends any packets to the attacker, and that the attacker therefore only ever receives packets from the zombie, since the victim and zombie only ever reply to packets using their real IP address as the return address.

Our goal is to ensure that the network satisfies the *non-interference property*, which is specified as: for any possible sequence of packets that the attacker can send to the victim and zombie, the sequence of packets the attacker receives in response is identical regardless of whether the target victim port is open or closed. This models the desired behavior that the attacker cannot gain any information about the target victim's port.

We do this by modeling two possible scenarios faced by the attacker which the attacker is attempting to distinguish and thus gain information about the victim. In each scenario, there is a victim and a zombie whose behavior and initial state are identical (except for the status of the target port on the victim, of course), but whose behavior and internal state over time can differ between scenarios through certain sequences of events due to the port status of the target port. In one scenario, the target port of the victim is open whereas in the second scenario, the target port is closed. The attacker sends identical packets in both scenarios.

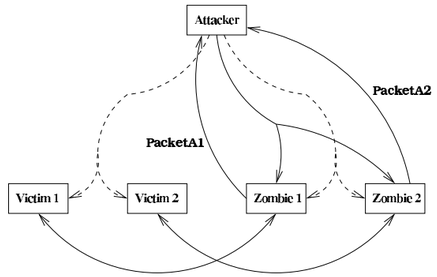



Figure 2: **Overview of our model (the IP address with no host that drops all packets is excluded from this figure for clarity).**

Figure 2 gives an overview of our model for testing non-interference properties of network stacks for idle scans. The status of the target port being open or closed is modeled as two different scenarios. Victim 1 and Zombie 1, for example, exist in scenario 1 where the target port on Victim 1 is open. Victim 2 and Zombie 2 exist in scenario 2 where the target port on Victim 2 is closed. The attacker can forge any arbitrary sequence of packets, but it must forge identical packets in both scenarios. The hosts in the different scenarios can respond differently and contain different internal state. `PacketA1` and `PacketA2` are the sequence of packets the attacker receives in scenario 1 and scenario 2, respectively.

In our model, the attacker can nondeterministically choose any arbitrary sequence of packets that do not violate the definition of an idle scan. Furthermore, the attacker need not reply to packets; the fact that the model allows the attacker to send any arbitrary packet covers all possibilities for reply. For the destination and return IP addresses of a packet, the attacker can choose among its own IP address, that of the victim or the zombie, or an IP address with no live host (that simply drops all packets). The only constraint is that the attacker cannot send a packet to the victim with its own IP address as the return IP address, as this violates the definition of an idle scan.

The attacker can distinguish between SYN cookies and regular SYN/ACKs that it receives in our model. This is true in reality due to the statistical properties of the initial sequence numbers of SYN cookies and the fact that they are never retransmitted whereas regular SYN/ACKs are.

The attacker can choose any values for the IP protocol (TCP, UDP, or ICMP), TCP flags, source and destination ports, validity of checksums, and so on. Every packet that the attacker forges is forwarded to the appropriate host in both scenarios.



### 3.2 SAL for modeling, generating counterexamples, and verifying properties

We model the network stack as a transition system. At an informal, high level, a transition system specifies computation as a sequence of transitions in a state machine. A state is given by the values of the local variables used to describe transitions. A transition system has an initial state. For every transition, there is an optional guard, which when true in the current state, leads the computation from the current state to the next state. For a nondeterministic transition system, multiple transitions may be triggered and one transition is randomly selected. This is repeated and the computation terminates if no guard is true in the current state. Dijkstra's guarded command language [7] is an example of a formalism for specifying transitions.

We used SAL (Symbolic Analysis Laboratory) for specifying the transition system and analyzing its properties. SAL is a language and a tool kit for specifying transition systems and analyzing them using model checking. SAL provides support for a suite of tools which have been successfully used for analyzing protocols and distributed algorithms (see [28]).

Figure 3 shows the outline of our SAL code for the model. Ellipses indicate where detailed code has been omitted, the full model is 895 lines of SAL code.

For a transition step, a nondeterministic choice is made between the attacker, victim, or zombie. If the attacker is chosen, it forges a nondeterministic packet, which can be a "drop" packet that has no effect. This packet is placed in the receive queue of the destination IP address. If the victim or zombie is chosen, it removes the next packet from its FIFO receive queue and replies based on its internal state and configuration. The functions `ProperReply` and `UpdateSyncache` are responsible for choosing the packet to reply with, if any, and any updates to the host's internal state (specifically the RST counter and SYN cache). Note that these are pure functions and do not update any state themselves.

Figures 4 and 5 show how the `ProperReply` and `UpdateSyncache` functions are used. A transition has a guard, *e.g.*, "`(z1.fullness /= 0 AND z2.fullness /= 0 ) -->`", which is a quantifier-free formula specifying a condition on the current state and must hold before the transition is executed, and then a formula relating the current state with the next state. An example of such a formula is "`z1'.fullness = z1.fullness - 1`", where `z1'` is the variable in the next state and `z1` is the variable in the current state. In this example the variable will be decremented by 1 in the next state.

When the guard on a transition for the zombie or victim fires, that host must remove a packet from its queue and then reply and update its state in both scenarios. `UpdateSyncache` returns not only the new state of the SYN cache, but also a variable called `.synPIsThere` which can take on the values `put`, `notexist`, or `exist`. This return value is passed to `ProperReply`, which needs to know if a SYN packet was put in an entry in the SYN cache, no entry was found for it because the SYN cache is full, or it already existed in the SYN cache. In this way `ProperReply` knows whether to send a SYN/ACK, send a SYN cookie, or drop the packet, respectively, if the packet is a SYN.

For example, if the zombie in one of the scenarios receives a SYN packet, it calls `UpdateSyncache` to determine the new state of the SYN cache and what will happen to the packet. If the internal state of the zombie indicates that there is a free entry in the SYN cache, the fact that the SYN will be placed in the SYN cache and the new status of the SYN cache are returned by this function. Then `ProperReply` is called with this information as an argument, and this function will determine that the proper reply is a normal SYN/ACK, with the destination IP address as the source of the SYN, valid checksums, *etc.*

Another example is that a host (a victim or zombie) receives a SYN/ACK. `UpdateSyncache` returns the current state (*i.e.*, no changes will be made to the SYN cache state) and then `ProperReply` will be called and will ignore `.synPIsThere` because the packet is not a SYN. The return value of `ProperReply` depends on the RST counter. If the RST counter is nonzero the return value of `ProperReply` will be a RST packet that the zombie will use for a reply and a reduced RST counter. If the RST counter is already zero, `ProperReply` will return a drop packet and zero still for the RST counter. All possible TCP, UDP, or ICMP packets and their corresponding replies are enumerated in `ProperReply` based on the reply that a typical network stack would send.

By forcing the zombies or the victims in both scenarios to reply at the same time step, the model stays in sequence. If a host in one scenario (*e.g.*, the victim in the closed port scenario) replies to a packet whereas the corresponding host in the other scenario (*e.g.*, the victim in the open port scenario) drops the packet, a "drop" entry is inserted in the destination host's queue as a filler and eventually ignored. This ensures that the packets that are received by the attacker vary only when non-interference is violated, *i.e.*, only when the sequence diverges.

SAL supports a suite of tools; the ones most relevant for the analysis discussed in this paper include a deadlock checker, a symbolic model checker for finite state systems based on the CUDD BDD package, and a bounded model checker based on the Yices SAT solver. Properties of a transition system are specified in Lin-

```
Network_Protocol_Stack : CONTEXT =
BEGIN
   % Type Declarations
      Protocol : TYPE = {tcp, icmp, udp, invalidProtocol};
      IP: TYPE = [1..5]; % 1 is Attacker, 2 is Victim, 3 is Zombie
      PortStatus: TYPE = {open, closed, filtered};
      TCP_Flag : TYPE = {syn, fin, synack, rst, drop};
      Port: TYPE = [1..3]; % each host has 3 ports
      Type_ICMP: TYPE = [1..5];
      Packet: TYPE = [# sip_IP : IP, dip_IP : IP,  ihl_IP : ValidOrInvalid, ...
      Host : TYPE = [# ip : IP , portstatus : array Port of PortStatus, ...
...
   % Functions
      ProperReply(Packet, PortStatus, Rst_counter): Packet, Rst_counter;
        % This function returns the proper response to a given packet
        % and a reduced Rst_counter if a RST is sent.
...
      UpdateSyncache (Packet, Syncache): Syncache, SynPIsthereOrNot;
        % This function returns the new state of the SYN cache, which can change
        % for SYNs or RSTs, and a value indicating if the SYN already exists, was
        % dropped, or was placed in the SYN cache
...
     main : MODULE =
       BEGIN
        % Variable Declarations
          GLOBAL v1 : Host % victim 1
          GLOBAL v2 : Host % victim 2
          GLOBAL z1 : Host % zombie 1
          GLOBAL z2 : Host % zombie 2
...
          LOCAL PacketA1 : Packet % Packets sent to attacker in scenario 1
          LOCAL PacketA2 : Packet % Packets sent to attacker in scenario 2
...
        % Initialization Section
        % Transition Section
        TRANSITION
          [
            (v1.fullness /= QueueSize OR z1.fullness /= QueueSize )-->
            % Attacker creates a packet and puts it in queues of either
            % v1 and v2 or z1 and z2
...
          []
            (v1.fullness /= 0 AND v2.fullness /= 0 ) -->
            % v1 and v2 pop a packet and call ProperReply and UpdateSyncache to
            % choose a proper reply and update their internal state.
...
          []
            (z1.fullness /= 0 AND z2.fullness /= 0 ) -->
            % z1 and z2 reply and update their state
...
          ]
       END;
     theorem1: THEOREM main |- G( PacketA1 = PacketA2 );
END
```

Figure 3: **Outline of SAL code for the network protocol stack model.**



```
...
[]
            (z1.fullness /= 0 AND z2.fullness /= 0 ) -->
            % z1 and z2 reply and update their state
...
            collector_1'.packet = z1.queueOfHost[1] ;
            (FORALL (j: FullnessOfQueue ): z1'.queueOfHost[j-1] = z1.queueOfHost[j]);
            z1'.fullness = z1.fullness - 1;
...
            temp'= UpdateSyncache(...);
            z1'.syncache= temp'.host.syncache;
...
            response' = ProperReply(collector_1', z1.portsstatus[collector_1'.packet.dport],
                          temp' ) ;
...
            % Do the same for z2, which may have a different packet in its queue
            collector_2'.packet = z2.queueOfHost[1] ;
            (FORALL (j: FullnessOfQueue ): z2'.queueOfHost[j-1] = z2.queueOfHost[j]);
            z2'.fullness = z2.fullness - 1;
...
```

Figure 4: **Overview of what happens when a transition fires.**

```
...
   % Functions
      ProperReply(Packet, PortStatus, Rst_counter): Packet, Rst_counter;
        % This function returns the proper response to a given packet
        % and a reduced Rst_counter if a RST is sent.
IF ...
...
       ELSIF ( Packet.protocol_IP = tcp AND  (Packet.seqNum_TCP = known AND Packet.ack_TCP = known )
         AND Packet.flag_TCP = syn AND ps = open AND Packet.synPIsThere2 = put  )
       THEN
            packetOut with .packet.sip_IP := Packet.dip_IP
                      with .packet.sport := Packet.dport
                      with .packet.dip_IP := Packet.sip_IP
                      with .packet.dport := Packet.sport
                      with .packet.flag_TCP := synack
                      with ...
...
      UpdateSyncache (Packet, Syncache): Syncache, SynPIsthereOrNot;
        % This function returns the new state of the SYN cache, which can change
        % for SYNs or RSTs, and a value indicating if the SYN already exists, was
        % dropped, or was placed in the SYN cache
       IF ( ... AND Packet.flag_TCP = syn AND Host.synQeueEntries[1]=valid
                    ... AND Packet.sip_IP = host.syncache[1].sip_IP ...)
       THEN  # Ignore duplicate SYNs, i.e., that already exist in the SYN cache
            synCollOut with .synPIsThere := exist
       ELSIF ( ... )
       THEN synCollOut
              with .synPIsThere := put
              with .host.syncache[1] := synCollIn.packet
...
```

Figure 5: **Structure of the `UpdateSyncache` and `ProperReply` functions.**

ear Temporal Logic (LTL). Our analysis involved using properties of the form $G(\alpha)$, where $\alpha$ is a quantifier-free, modality free formula expressed using state variables, to mean that $\alpha$ holds in every state of the transition system. The non-interference property is specified as:

$$\vdash \mathrm{G}(\mathrm{PacketA1} = \mathrm{PacketA2})$$

This means that the sequence of packets the attacker receives in response to its probes from the zombie in the first scenario is always identical to the response from the zombie in the second scenario.

We have used SAL's bounded model checker for finding counterexamples as it is depth-first and explicitly enumerates states. SAL's symbolic model checker, which is exhaustive, is useful for finding smaller counterexamples as well as for proving properties of interest, which are often difficult to do by explicit state enumeration model checkers. A useful comparative study of exhaustive symbolic model checkers and explicit state enumeration model checkers is in [5] for protocol analysis and controllers.

### 3.3 Assumptions to reduce the number of states in the model

A number of assumptions were made to keep our model simple. Our strategy was to start with a simple model and introduce additional complexity into the model if no counterexamples are generated, and ensure that the abstractions we made caused no loss of generality that would exclude potential counterexamples.

A major abstraction in the model is that we consider the proper reply to SYN/ACK packets to be "drop" for open ports and RST for closed ports. In reality, network stacks that respond differently to SYN/ACKs on open *vs.* closed/filtered ports typically respond with RSTs or ICMP and have different rate limits per port. Since the lower rate limit (typically ICMP) will cause drops before the higher rate limit, without loss of generality, we can consider open ports to simply drop SYN/ACK packets from the initial state. This is equivalent to assuming that the attacker immediately exhausts the lower rate limit.

We also exclude ICMP and UDP from the split SYN cache version of our model. Since ICMP host error packets have the same effect on the SYN cache as RSTs, and other ICMP and UDP packets make no relevant changes to the destination host's TCP state, ICMP and UDP do not affect the non-interference property for the SYN cache structure. Invalid checksums in packet headers are also excluded, because they are dropped without affecting the state of the destination host in all cases.

Another major abstraction is that each of the two buffers in our split SYN cache has only a single entry. There are three reasons why only a single entry in the SYN cache is necessary in the model:

- Pending entries in the SYN cache with source IP addresses and ports (possibly forged by the attacker) that correspond to invariant ports (that have the same status in both scenarios) cannot cause divergence in the internal state of any of the hosts in the two scenarios. Thus, no more than one such SYN cache entry at a time can be useful for creating a counterexample.
- Even though RST rate limiting is performed separately for open and closed ports, the rate limit value stored by any host cannot be caused to diverge on invariant ports. Only the target port on the victim can cause divergence. Since only one such port exists, only one SYN cache entry at a time can be useful for creating a counterexample. If we had not received a counterexample under this assumption, we would have incrementally allowed more entries in the SYN cache.
- While the single entry is full, the SYN cookies generated in response to dropped SYN packets can only cause internal state differences if sent to the target port on the victim. If this is the case then the entry in the SYN cache cannot also be a SYN packet with the victim IP address and target port, since duplicate SYNs are ignored by the SYN cache. Thus, one SYN cache entry per trust level (trusted and untrusted) is general enough to handle all of the cases of any number of SYN cache entries.

Because of the above simplification of making the SYN cache have a single entry for each trust level, we modeled only three ports without loss of generality. Port 1 is prohibited (*e.g.* by a firewall rule) to the attacker for the split SYN cache implementation, meaning that the attacker cannot send packets to port 1. This was done so that we could include the RST rate limitation, which has important interactions with the SYN cache, without receiving the RST rate limit counterexample. Port 1 is closed in both scenarios for the zombie; however, for the victim, it is open in one scenario and closed in the other. In other words, port 1 is the target port for the attacker to get information. Port 2 is closed and port 3 is open on both hosts in both scenarios. Because closed ports are equivalent in terms of their responses, a single closed port per host is equivalent to any number of closed ports. Because the SYN cache has a single entry and open ports only have different behaviors based on the status of the SYN cache, a single open port per host is also equivalent to any number of open ports.

In real SYN cache implementations, there is a timeout after which SYNs that have not become fully open TCP



| Port | Zombie status | Victim Status |
|---|---|---|
| 1 | Open | Open in scenario 1, closed in scenario 2 |
| 2 | Closed | Closed |
| 3 | Open | Open |

Table 1: **Ports and their status in our model.**

connections are dropped. Because our model allows the attacker to remove any entry from the SYN cache at any time via a RST packet (which is also possible in reality for Linux SYN cache implementations), our model need not incorporate this timeout. Also, RST rate limiting is done per a time period in reality. A fixed limit of RSTs for an unbounded amount of time is a generalization of this that does not exclude any counterexamples because for any violation of non-interference based on a rate limit a single time period is enough to create a counterexample.

## 4 Finding and ameliorating idle scans

In this section, we describe the counterexamples that our modeling effort produced and give experimental results of an implementation of these counterexamples to demonstrate that they can indeed be used to do idle port scans.

### 4.1 Discovering counterexamples

We now describe the two counterexamples that were discovered during the process of developing the model.

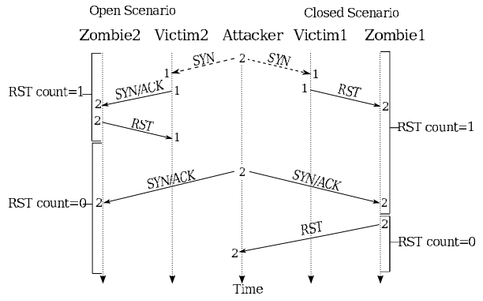



Figure 6: **RST rate limiting counterexample.**

#### 4.1.1 RST rate limiting counterexample

When we applied SAL's bounded model checker to a simpler version of the model, in which the SYN cache did not play any role, for the property "⊢ G(PacketA1 = PacketA2)" SAL identified a counterexample with `RST_counter` set to 3 in the initial state. We simplified the model further by reducing the initial value of `RST_counter` to 1 and still received a counterexample. The counterexample in this case was found much more quickly, at depth 5 in the transition system.

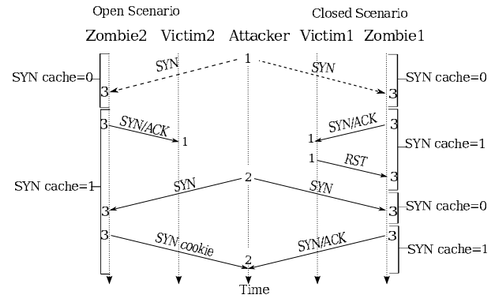



Figure 7: **SYN cache counterexample.**

This counterexample is illustrated in Figure 6. The figure shows the sequence of packets for the open *vs.* closed scenarios that that attacker can send to distinguish between the scenarios. Note that the attacker sends the same sequence of packets in both cases. Dashed lines are forged packets (for Figure 6 the packets are forged so that they appear to come from the zombie). The numbers at the bases and heads of the arrows represent the source or destination port number, respectively. The RST count state for a period of time is the state of that variable for each scenario.

In this example the attacker wants to discern the port status (open or closed) of port 1 on the victim. Port 2 on the zombie is closed. The packets the attacker sends are identical in both scenarios. The attacker cannot see the packets that are sent between the victim and zombie or zombie and victim. The port status must be inferred by the difference in the expected packet sequence that the attacker will see between the two scenarios. First, the attacker forges a SYN to the victim on the target port that appears to be from the zombie with return port 2. If the target port on the victim is open, it will respond to the zombie with a SYN/ACK on the zombie's closed port 2, causing the zombie to send the victim a RST and decrement its RST count. If the port is closed, the victim will respond to the zombie with a RST which the zombie ignores. Next, the attacker sends a SYN/ACK packet, using its own return IP address, to the closed port on the zombie. If the attacker receives a RST in response, then it can infer that the victim target port status is closed since an open port would have caused the zombie to have already reached its RST rate limit.

#### 4.1.2 SYN cache counterexample

For the second case, we tried a more complex model that included a SYN cache. We started with a SYN cache of size 2, then simplified it further to size 1, and SAL's bounded model checker still identified the counterexample to the non-interference property as illustrated in Figure 7.

The relevant state in this case is the number of pending SYN/ACK entries in the SYN cache, with a maximum value of 1 in our model. The notable thing about this form of idle scan is that the attacker never sends any packets to the victim, not even packets with forged return IP addresses. Instead, the attacker sends a SYN to the zombie on an open port, with the return IP address of the victim and the return port as the target port. The zombie places this SYN packet in the SYN cache, which in our model has only a single entry, and sends a SYN/ACK response to the victim. If the victim target port is closed it will send a RST in response, which causes the zombie to remove the relevant SYN cache entry so that there is now a free entry in the SYN cache. An open target port on the victim will simply drop the SYN/ACK from the zombie, so that the SYN cache of the zombie remains full since the zombie is still waiting for a response to the SYN/ACK. The attacker can then infer the status of the zombie's SYN cache, and therefore the victim port status, by sending a SYN to the zombie with the attacker's own return IP address. A regular SYN/ACK means the SYN cache entry was free, a SYN cookie indicates that it was full.

Note that responses to SYN/ACKs on open, closed, or filtered ports vary for different operating systems, but all that matters is that for open *vs.* closed or open *vs.* filtered the response differs in some way under certain conditions. More discussion of the possibilities for this is in Section 4.2. The SYN cache counterexample makes it possible to, *e.g.*, port scan a network on a port that is blocked for the entire network from outside the firewall. Imagine in Figure 7 that the zombie and victim are behind a firewall and the attacker is outside the firewall. Even if the firewall drops all incoming packets with destination port, *e.g.*, 22 for Secure Shell (SSH), the attacker can scan port 22 on the network by using other open ports. Also, there may be firewall rules that enforce that only trusted machines (*e.g.*, the zombie) can route packets to the victim. In this case the victim might be an internal database server and the zombie is the web server interface to the database, for example. Information about what ports the victim has open might give the attacker an idea of whether compromising the zombie to subsequently get access to the victim is worth the effort and risk. It might also be that the attacker can route packets to the victim, but not on the target port. For example, many machines leave certain ports open only for their backup servers that contact them nightly. Or, the system administrator might only allow incoming SSH sessions on their critical servers from their own office machines and not from other IP addresses. Knowing these kinds of trust relationships and exploiting them to find out more about the victim machines can be very valuable to an attacker.

For each host, both the SYN cache and the reset rate limiting variables constitute shared, limited resources, which are the sources of violations of non-interference in our two counterexamples.

### 4.2 Experimental confirmation of counterexamples

We implemented both counterexamples to verify that these two new forms of idle scan that resulted from the modeling effort were possible for real hosts. Our results presented in this section demonstrate that the differences in the sequence of packets the attacker sees translate from the abstract notion of non-interference in our model to differences that can be seen in real network packet traces. Our implementations of the two idle scans are not optimized for speed or stealth, nor do they account for packet loss or packet delay, but in this section we discuss the practicality of these two forms of idle scan and conclude that they are both practical.

#### 4.2.1 Experimental setup

For our experiments, we set up VirtualBox [34] virtual machines connected using IPv4 on two different subnetworks with TUN/TAP interfaces. The attacker machine was the host, and one subnet contained a Linux kernel 2.4 host (Fedora Core 1) which served as the zombie for the SYN cache idle scan implementation. The other subnet contained a Windows XP host with no service packs, a Linux kernel 2.6 host (CentOS 5.2), and a FreeBSD 7.1.1 host. The latter served as the zombie for RST rate limiting idle scan implementation. IP forwarding between these two subnetworks was performed by the host. Packets were generated and captured by separate threads using the Perl Net::RawIP and Net::Pcap libraries, respectively.

#### 4.2.2 RST rate limiting idle scan implementation

In our transition system model, RSTs are limited to a finite number for infinite time. For a real FreeBSD system, RSTs are limited to a default of 200 per second, with separate limitations for open and closed ports. Our implementation sends 2000 each of two different types of packets, each at a rate of 180 per second, to the victim and FreeBSD zombie, respectively. One type of packet is forged SYNs to the victim on the target port that appear to be from the zombie on a port that is closed on the zombie. The other is SYN/ACKs from the attacker to the zombie, which the zombie should reply to with RSTs. If the zombie is sending RSTs at a rate of 180 per second to the victim in response to the victim's SYN/ACKs (meaning the victim target port is open), this should interfere with the rate at which the zombie sends RSTs to the attacker. Thus the number of RSTs the attacker receives in our experiment can be used to infer the port status of the target port on the victim. We repeated the RST rate limiting idle scan experiment 700 times each for an open and closed port on the victim. The victim was a Linux kernel 2.6 virtual machine. The host-based firewalls on both machines were disabled, although for the victim the idle scan works whether the host-based firewall is enabled or not. For FreeBSD, RST rate limiting does not apply to filtered ports. The pf host-based firewall is disabled by default for FreeBSD installations.

| Port status | Mean | Std. dev. | Min. | Max |
|---|---|---|---|---|
| Open | 1552.1 | 47.0 | 1429 | 1634 |
| Closed | 2000 | 0 | 2000 | 2000 |

Table 2: **Results from RST rate limiting idle scan implementation.**

The results from our RST rate limiting idle scan are shown in Table 2, where the results are based on the number of RSTs the attacker receives. When the victim port is closed, the attacker receives all 2000 RST responses from the zombie. When the victim port is open, the attacker receives at most 1634 RSTs. Thus, determining if the target port is open or closed is straightforward for idle scans based on RST rate limiting.

#### 4.2.3 SYN cache idle scan implementation

SYN cache implementations vary for different operating systems. While the SYN cache idle scan is possible using virtually any host as a zombie, the simplest network stack to use as a zombie is Linux kernel 2.4. Linux kernel 2.4 uses a simple buffer for the SYN cache, with between 128 and 1024 entries depending on the memory available on the system. Our Linux kernel 2.4 virtual machine zombie had a SYN cache size of 128, but Linux enforces a rule that only three fourths of the SYN cache can contain SYN packets from hosts that have not demonstrated their liveness in the recent past by completing a fully open TCP connection. This effectively reduces the SYN cache size to 97. We did not enable SYN cookies, which are disabled by default in Linux. The attack works basically the same whether or not SYN cookies are enabled. We ran two separate sets of experiments for the SYN cache idle scan implementation, one to demonstrate that it is possible for the attacker to detect the presence of live machines and perform a rudimentary form of operating system detection, and another to demonstrate that under certain circumstances the attacker can infer the port status of a target port on a particular victim IP address. For all experiments, 100 data points were generated for both open and closed port scenarios.

For checking for liveness, we scanned four different IP addresses. One is a default FreeBSD 7.1.1 machine (with the pf host-based firewall disabled, as is the default), another is a Windows XP machine with no service packs (with Windows firewall disabled, as is the default), and a third is a Linux kernel 2.6 machine (CentOS 5.2, with iptables enabled, as is the default). The fourth IP address has no live host so all packets are simply dropped. Forging packets from random return IP addresses on these victims is very likely to send SYN/ACKs to closed or filtered ports, so we choose random return ports for all forged SYNs where the attacker uses the victim as the return IP address. Varying this return port number is important because if the return port is not different then the forged SYNs will have the same IP addresses and ports for both the destination and source and the SYN cache will drop such duplicates. Both RSTs and ICMP errors cause their corresponding entries to be removed from the SYN cache when received by the zombie.

Because Linux responds to SYN/ACKs on filtered ports at a very low rate (about 10 per second) with ICMP host prohibited packets, FreeBSD responds to SYN/ACKs on closed ports at a rate of at most 200 per second, Windows responds on closed ports with RSTs at an unlimited rate—and IP addresses without live hosts simply cause SYN/ACKs to be dropped—it is possible for the attacker to idle scan a subnetwork and infer something about the operating systems that the live hosts discovered have installed. To scan a single IP address, our implementation sends 50 forged SYNs (that appear to be from the victim), then 50 each of forged SYNs and SYNs where the attacker uses their own return IP address, and then 200 more forged SYNs, all at a rate of 1000 per second. It then sends 200 each of forged SYNs and SYNs where the attacker uses their own return IP address at a rate of 400 per second. The number of SYNs where the attacker uses their own return IP address and receives a SYN/ACK response can then be used to infer the liveness and operating system of the IP address. The results from this experiment are shown in Table 3, where the results are based on the number of SYN/ACKs the attacker receives (note that for Linux kernel 2.4 network stacks SYN/ACKs are retransmitted five times until they time out after 190 seconds).

Under certain circumstances, it is also possible to port scan specific ports on a particular IP address using a SYN

| Host | Mean | Std. dev. | Min. | Max |
|---|---|---|---|---|
| Not live | 109.1 | 7.4 | 96 | 123 |
| Linux 2.6 | 126.9 | 6.3 | 111 | 138 |
| FreeBSD | 300 | 0 | 300 | 300 |
| Windows XP | 460.3 | 11.9 | 435 | 477 |

Table 3: **Results from SYN cache idle scan implementation for liveness and operating system.**

cache-based idle scan. Specifically, if the response or rates differ for open *vs.* closed or filtered ports on the victim then scanning a target port is possible. Examples of this are FreeBSD with the pf host-based firewall disabled, where open ports and closed ports are rate-limited separately, or Linux hosts with the iptables host-based firewall enabled and an open port that does not use the stateful module of iptables.

To test the FreeBSD example, we developed a SYN cache-based idle scan that simultaneously sends 20000 forged SYN packets (with random return ports that are closed on the zombie) as quickly as possible while sending, at half the rate, alternating forged SYNs with the target port on the victim as the source port and valid SYNs with the return address of the attacker. Because closed ports on the victim are rate limited due to the forged SYNs with random return ports coming from the zombie, the forged SYNs with the target port on the victim as their return port will quickly fill the SYN cache if the target port is also closed and cause fewer entries to be free for non-forged attacker SYNs, therefore causing the attacker to see fewer SYN/ACKs in response. If the target port is open, the open port sends more RSTs before rate limiting begins meaning that more SYN cache entries remain free and the attacker sees more SYN/ACKs. The results of this experiment are shown in Table 4, where the results are based on the number of SYN/ACKs the attacker receives. Some data points for both closed and open ports were thrown out due to failures of the Python pcap library at high packet rates. Packet loss due to the high rates could only make the distributions more similar, not less, because more packets are sent over the TUN/TAP interface for the open port scenario. Thus, the distributions for open and closed ports are clearly different. A two-sampled, unpooled $t$-test (which assumes neither known variances nor equal variances) for these two sets of data gives a $t$ score of 7.71 with 197 degrees of freedom, which corresponds with a $p$-score of 0.999999999999696 meaning that a null hypothesis that the two distributions have an equal mean is rejected with very high confidence.

For port scanning Linux-based victims, the idle scan first sends 96 filler SYNs to fill all but one entry in the SYN cache. SYN/ACK replies to filler SYNs are not counted in the results. Then it alternates, at an overall

| Port status | Mean | Std. dev. | Min. | Max |
|---|---|---|---|---|
| Open | 262.1 | 41.8 | 120 | 447 |
| Closed | 218.0 | 39.3 | 68 | 318 |

Table 4: **Results from SYN cache idle scan implementation for port scanning FreeBSD.**

| Port status | Mean | Std. dev. | Min. | Max |
|---|---|---|---|---|
| Open | 482.4 | 3.3 | 474 | 489 |
| Closed | 427.8 | 3.4 | 417 | 435 |

Table 5: **Results from SYN cache idle scan implementation for port scanning Linux.**

rate of 100 packets per second, forged SYNs with the return IP address of the victim and return port of the target port, filler SYNs, and probe SYNs. Table 5 shows the results of these experiments, where the results reflect the number of SYN/ACK responses to probe SYNs.

#### 4.2.4 Stealth and efficiency

Our idle scan implementations in this section are intended to show that the abstract counterexamples that resulted from our modeling effort were real divergences in real network stacks that could be exploited by the attacker for idle scans. Since the divergences are based on rates in real network stacks we used hypothesis testing to show this. We only report a $t$-score and $p$-score for one set of experiments (the SYN cache idle scan implementation for port scanning FreeBSD) because the distributions of the results for other experiments were so different that their high $t$-scores led to $p$-scores that were within floating point rounding error of 1.0. Our implementations of these idle scans were designed for this hypothesis testing and therefore are not optimized for attacker stealth or efficiency in carrying out the scan. For assessing the practicality of these idle scan techniques, we will now comment on stealth and efficiency.

For the RST rate limiting idle scan, the attacker cannot perform the idle scan without sending more than 200 SYN/ACKs to the zombie either directly or indirectly. However, the attacker need not send SYNs to the victim (forged from the zombie) at half this rate. It is possible to, *e.g.*, send SYNs to the victim at a rate of 20 per second and send SYN/ACKs (or any packet that will elicit a RST) to the zombie at 195 per second. Theoretically, the mutual information between the victim port status and the sequence of packets the attacker sees is non-zero even if the attacker sends only a single forged SYN to the victim, and even when packet loss is accounted for. Thus the attacker has a fair amount of flexibility in terms of trading off speed of the scan *vs.* stealth for packets it sends to the victim. Furthermore, sending SYNs simultaneously to multiple victims and multiple ports and measuring the zombie responses in the aggregate can increase the efficiency of the scan if the distribution of expected closed *vs.* open victim ports diverges from an equal distribution. To see this, consider the extreme case where a large subnetwork has only a single host with an open port, something similar to a binary search could greatly reduce the amount of time necessary for the scan in this case.

For the SYN cache idle scans, which are more powerful in terms of the new capabilities they offer attackers beyond the currently known idle scan technique, there is a wider range of efficiency and stealth tradeoffs that the attacker can make. Furthermore, unlike ICMP IPID- or RST rate limit-based idle scans, virtually any modern network stack that offers any type of protection against SYN flooding can be used as a zombie. We chose to use a low-memory Linux kernel 2.4-based zombie for our experiments due to its simplicity and small SYN cache size, but larger SYN cache sizes or more complex SYN cache implementations are also easily exploited for SYN cache idle scans. The SYN cache only needs to be almost full for SYN cache idle scans to work, and SYNs for half-open connections take 190 seconds to timeout in Linux by default. So even for high-memory Linux 2.4 machines with 1024 SYN cache entries (of which 769 are used, compared to 97 for 128-entry SYN caches), the rate necessary to create the conditions for an idle scan only increases from 0.5 SYNs per second from the attacker to the zombie to about 4.1 SYNs per second (these rates keep the buffer almost full despite the timeouts). Once these conditions are created, the attacker effectively can do a SYN/ACK scan of the victim host or network at the cost of two packets sent per SYN/ACK query and three more generated as responses. It also does not matter whether or not the zombie implements SYN cookies, since SYN cookies are never retransmitted (compared to typically three to five retransmissions of regular SYN/ACKs for various zombie configurations) and also have easily identifiable statistical anomalies in their initial sequence numbers.

Some SYN cache implementations that are not simple buffers like Linux 2.4 may make SYN-cache idle scans slightly more difficult, but still possible and relatively efficient. For example, the FreeBSD SYN cache implementation [19] uses a SYN cache with 512 buckets that each have 30 entries and are chosen uniformly at random using a hash of the IP address/port pairs and a host-generated secret. This mechanism is designed to stop denial-of-service, not idle scans. It creates some equivocations that can reduce the amount of information flow the attacker can exploit for idle scans but the attacker can still perform the scan relatively efficiently even with FreeBSD zombies. We have not explored the SYN cache implementations of Linux kernel 2.6 or Windows hosts, but all modern network stacks must have some form of SYN cache for reliability purposes and a limit on this resource to prevent denial-of-service. Thus, only by making this resource non-shared is non-interference to prevent idle scans possible, and the current OSI network stack model with TCP/IP does not make the necessary trust distinctions to split the SYN cache. Thus, virtually every end host machine that the attacker can route to at least one open port on is a potential zombie.

The rate at which the attacker must send packets to the zombie for a SYN cache idle scan, and therefore the stealth of the scan, depends on the attacker's goals. If the zombie is a Linux kernel 2.4 machine and the attacker wants only to check the liveness of a range of IP addresses on the victim network, then between 0.5 and 4.1 packets per second plus the probes themselves is sufficient. Note that, in terms of stealth, it is also relevant that the attacker need not send any packets to the victim for this form of idle scan, not even packets with forged return addresses.

For detecting the operating system of a victim host or scanning individual ports on the victim, higher rates are necessary. Detecting a Linux machine on the victim network and port scanning it can easily be done at between 10 and 20 packets per second. We also discovered during our experiments that, at least for Linux kernel 2.4 hosts, it is easy for the attacker to not only remove their own packets from the SYN cache manually, but any packet that they have forged, using forged RSTs. This is because only the IP address and port pairs are checked, the sequence and acknowledgment numbers for RSTs are ignored when deciding whether to drop an entry from the SYN cache on the zombie. Thus, the attacker has a high degree of control over the SYN cache status of the victim. Packet delay, packet loss, and interference from other machines that contact the zombie can easily be accounted for in this way, and the aggregate effect of scanning multiple victims at a time mentioned above also applies to SYN cache idle scans. In future work we intend to model the capabilities of this attack as a Markov Decision Process and discern tight bounds on numbers of packets and rates needed for different attacker goals.

In terms of the practicality of our attacks, the ability to scan firewalled ports and discover machines on protected networks that the attacker cannot route packets to certainly underscores the need for good ingress filtering and DMZ management. Our attacks are applicable in all three of the following scenarios: when the victim and zombie are on the same subnet and communicate using ARP[2], when the victim and zombie are within the same network domain but on different subnets, and when the victim and zombie are geographically separated by some distance on the Internet. The attacker can be inside or outside the network domain of the zombie and victim. Thus, many possibilities for network inference

arise. For example, the attacker can infer when a host opens ports only to other particular machines, such as a backup server or network administrator. While many network configurations prevent IP address spoofing, which is essential for both attacks described in this paper, idle scans are a very general technique that can apply in a variety of scenarios.

### 4.3 Ensuring non-interference using the SAL model checker

Based on our experimental results from implementing the two counterexamples as idle scan attacks, it is apparent that RST rate limiting and the SYN cache interact in complex ways and cannot be considered separately. Thus we chose to leave RST rate limiting in the model for verifying non-interference of the split SYN cache.

It is well-known that verifying properties using a model checker is much more difficult than finding a counterexample. We abstracted the model down to the simplest form that produces both counterexamples, and attempted to prove the non-interference property for cases where the shared, limited resources were split based on trust relationships and therefore no longer shared. The zombie and victim consider each other trusted and the attacker untrusted. For the RST rate limiting counterexample, the hosts have separate `RST_count` RST counters for trusted *vs.* untrusted hosts, and the SYN cache is removed. For the SYN cache counterexample, we implemented a split SYN cache structure with separate SYN cache buffers for trusted *vs.* untrusted hosts.

In the first case, we removed the SYN cache and focused only on the RST rate limitation counter example. Since symbolic model checkers are known to be better for verifying properties in contrast to explicit state enumeration based bounded-model checkers, we used SAL's symbolic model checker. It verified the property that:

$$\vdash \mathrm{G}(\mathrm{PacketA1} = \mathrm{PacketA2})$$

This verification completed in a little over 5 minutes.

Encouraged by this result, we introduced the SYN cache back into the model. The symbolic model checker ran out of memory on a machine with 16GB of memory after three days. We then ran the bounded model checker up to depth 1000 (to mean that all sequences of transitions of length $\leq 1000$ are checked for counterexamples), and the model checker did not report any counterexample, which is very encouraging. This means that the attacker cannot violate non-interference with any idle scan where less than 1000 transitions occur. The SYN cache counterexample to our shared SYN cache implementation required only 5 transitions. Informally, this result means that there exists no attack, even with only a single entry in the SYN cache, where the attacker can violate non-interference with 1000 or fewer packets being generated by the attacker or by the zombie and victim's responses. Currently, we are exploring alternatives to symbolic model checking for the split SYN cache, including verifying the property through $k$-induction [6]. We are also considering attempting a proof by induction on an induction-based theorem prover such as ACL2 or RRL.

## 5 Concluding remarks and future work

We modeled idle scans for modern network stacks using transition systems and analyzed them using model checking. This modeling effort led to the discovery of two new forms of idle scan, each of which was associated with a shared, limited resource. Our results demonstrate that non-interference for network protocol stacks warrants further study. We discovered two new forms of idle scan, one of which gives the attacker capabilities that no current attacker port scanning capabilities below layer 7 (the application layer) provide. We demonstrated in this paper that it is possible for an attacker to port scan a network from outside the firewall on a port that the firewall blocks, for example. We also showed that this form of idle scan, based on SYN caches, can be used for a rudimentary form of operating system detection. In light of these results, a more formal treatment of information flow in networks is needed so that we can better understand advanced idle scans, both for existing networks and in future protocol designs.

We discussed the stealth and efficiency of the idle scans in Section 4.2.4. While it is clear both that the attacks are practical and that certain defenses exist in some situations, a more thorough treatment of possible scans and defenses to detect or eliminate them is needed. By modeling idle scans as a Markov Decision Process, it will be possible to explore this space more thoroughly and find boundaries in terms of packet rates.

Using SAL's model checkers, we were able to identify counterexamples to non-interference, in the form of idle scans, from our formal model of a network stack as a transition system. After fixing the model by splitting limited resources and separating them for trusted *vs.* untrusted hosts we are able to verify the non-interference property for the RST rate limit case. However, we were only able to show the non-interference property with both RST rate limiting and a SYN cache up to 1000 transitions. Verifying the non-interference property for this more general fix using a model checker remains a challenge. We plan to investigate induction methods for this. While non-interference and model checking proved useful for studying specific shared resources, we were not able to build a model complex enough to discover unexpected counterexamples.

Our model of network stacks was at the level of abstraction of sequences of packets. A richer model that includes memory usage, packet loss, and packet delay would likely produce more counterexamples to the non-interference property for idle scans. Thus we propose that trust relationships be made explicit all the way down to the IP layer in future protocol designs. Because all resources are inherently limited, giving protocol implementations a mechanism that can help divide these resources and remove sharedness is the only way to address the advanced network reconnaissance attacks of the future. Our results in Section 4.3 demonstrated that non-interference, which effectively eliminates idle scans, is achievable by statically dividing resources based on trust relationships.

## Acknowledgments

We are grateful to the anonymous reviewers for their suggestions. We would also like to thank many others who made suggestions about model checking tools and gave us helpful feedback on this research, and Stephanie Forrest for her support of Roya Ensafi. This work was supported in part by the U.S. National Science Foundation (CNS-0905177, CNS-0844880, CNS-0831462, CNS-0905222, CNS-0653951). Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

[1] ANTIREZ. new tcp scan method. Posted to the bugtraq mailing list. 18 December 1998.

[2] BELLOVIN, S. M. A technique for counting NATted hosts. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment* (New York, NY, USA, 2002), ACM, pp. 267–272.

[3] BENSALEM, S., GANESH, V., LAKHNECH, Y., NOZ, C. M., OWRE, S., RUESS, H., RUSHBY, J., RUSU, V., SAÏDI, H., SHANKAR, N., SINGERMAN, E., AND TIWARI, A. An overview of SAL. In *LFM 2000: Fifth NASA Langley Formal Methods Workshop* (Hampton, VA, jun 2000), C. M. Holloway, Ed., NASA Langley Research Center, pp. 187–196.

[4] BERNSTIEN, D. J. SYN Cookies. `http://cr.yp.to/syncookies.html`.

[5] CHOI, Y. From NuSMV to SPIN: Experiences with model checking flight guidance systems. *Form. Methods Syst. Des. 30*, 3 (2007), 199–216.

[6] DE MOURA, L. SAL tutorial. CSL technical note, Computer Science Laboratory, SRI International, Menlo Park, CA, Apr. 2004. Available at `http://sal.csl.sri.com/doc/salenv_tutorial.pdf`.

[7] DIJKSTRA, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM 18*, 8 (1975), 453–457.

[8] EDMUND, S. C., CLARKE, E. M., SHARYGINA, N., AND SINHA, N. State/event-based software model checking. In *Integrated Formal Methods* (2004), Springer-Verlag, pp. 128–147.

[9] GATES, C. *Co-ordinated port scans: a model, a detector and an evaluation methodology*. PhD thesis, Dalhousie Univ., Halifax, Canada, Canada, 2006.

[10] GATES, C. Coordinated scan detection. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 09)* (Feb. 2009).

[11] GOGUEN, J. A., AND MESEGUER, J. Security policies and security models. In *IEEE Symposium on Security and Privacy* (1982), pp. 11–20.

[12] GU, G., SHARIF, M., QIN, X., DAGON, D., LEE, W., AND RILEY, G. Worm detection, early warning and response based on local victim information. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 136–145.

[13] HAMED, H., AL-SHAER, E., AND MARRERO, W. Modeling and verification of IPSec and VPN security policies. In *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 259–278.

[14] JUNG, J., PAXSON, V., BERGER, A. W., AND BALAKRISHNAN, H. Fast portscan detection using sequential hypothesis testing. In *Proceedings of the IEEE Symposium on Security and Privacy* (2004).

[15] KANG, M. G., CABALLERO, J., AND SONG, D. Distributed evasive scan techniques and countermeasures. In *DIMVA '07: Proceedings of the 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 157–174.

[16] KEMMERER, R. A. Shared resource matrix methodology: an approach to identifying storage and timing channels. *ACM Trans. Comput. Syst. 1*, 3 (1983), 256–277.

[17] KOHNO, T., BROIDO, A., AND CLAFFY, K. C. Remote Physical Device Fingerprinting. *IEEE Symposium on Security and Privacy* (May 2005).

[18] LECKIE, C., AND KOTAGIRI, R. A probabilistic approach to detecting network scans. In *IEEE/IFIP Network Operations and Management Symposium. (NOMS).* (2002), pp. 359–372.

[19] LEMON, J. Resisting SYN flood DoS attacks with a SYN cache. `http://people.freebsd.org/~jlemon/papers/syncache.pdf`.

[20] LYON, G. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure.Org LLC, Sunnyvale, CA, USA, 2009.

[21] MCCAMANT, S., AND ERNST, M. D. A simulation-based proof technique for dynamic information flow. In *PLAS 2007: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (San Diego, California, USA, June 14, 2007).

[22] MCCAMANT, S., AND ERNST, M. D. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation* (Tucson, AZ, USA, June 9–11, 2008).

[23] MUELDER, C., LIU MA, K., AND BARTOLETTI, T. Interactive visualization for network and port scan detection. In *Proceedings of 2005 Recent Advances in Intrusion Detection* (2005), p. 05.

[24] PANG, R., YEGNESWARAN, V., BARFORD, P., PAXSON, V., AND PETERSON, L. Characteristics of internet background radiation. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2004), ACM, pp. 27–40.

[25] RITCHEY, R. W., AND AMMANN, P. Using model checking to analyze network vulnerabilities. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2000), IEEE Computer Society, p. 156.

[26] RUSHBY, J. SAL tutorial: The Needham-Schroeder protocol in SAL. CSL technical note, Computer Science Laboratory, SRI International, Menlo Park, CA, Oct. 2003. Available at `http://www.csl.sri.com/users/rushby/abstracts/needham03`.

[27] RUSHBY, J. SAL tutorial: Analyzing the fault-tolerant algorithm OM(1). CSL technical note, Computer Science Laboratory, SRI International, Menlo Park, CA, Apr. 2004. Available at `http://www.csl.sri.com/users/rushby/abstracts/om1`.

[28] SAL PROJECT. Examples. `http://sal.csl.sri.com/examples.shtml`.

[29] SCHECHTER, S., JUNG, J., AND BERGER, A. W. Fast Detection of Scanning Worm Infections. In *7th International Symposium on Recent Advances in Intrusion Detection (RAID)* (French Riviera, France, September 2004).

[30] SRIDHARAN, A., AND YE, T. Tracking port scanners on the IP backbone. In *LSAD '07: Proceedings of the 2007 workshop on Large Scale Attack Defense* (New York, NY, USA, 2007), ACM, pp. 137–144.

[31] SRIDHARAN, A., YE, T., AND BHATTACHARYYA, S. Connectionless port scan detection on the backbone. In *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International* (April 2006), pp. 10 pp.–576.

[32] STANIFORD, S., HOAGLAND, J. A., AND MCALERNEY, J. M. Practical automated detection of stealthy portscans. *J. Comput. Secur. 10*, 1-2 (2002), 105–136.

[33] SUTHERLAND, D. A model of information. In *Proceedings of the 9th National Computer Security Conference* (1986).

[34] VirtualBox. `http://www.virtualbox.org/`.

[35] WEAVER, N., STANIFORD, S., AND PAXSON, V. Very fast containment of scanning worms. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 3–3.

[36] WHYTE, D., KRANAKIS, E., AND VAN OORSCHOT, P. C. DNS-based detection of scanning worms in an enterprise network. In *Proc. of the 12th annual Network and Distributed System Security symposium* (2005), pp. 181–195.

[37] WRAY, J. C. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy* (1991), pp. 2–7.

[38] YUMEREFENDI, A., MICKLE, B., AND COX, L. P. TightLip: Keeping applications from spilling the beans. In *Networked Systems Design and Implementation (NSDI)* (2007).

## Notes

[1] These counterexamples were discovered during the process of deciding what details to include in the model, so resulted from the modeling effort but were not unexpected results from the model checker itself.

[2] We have confirmed that typical gateways and hosts handle ARP properly even with spoofed IP addresses.



# Building a Dynamic Reputation System for DNS

Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster
College of Computing, Georgia Institute of Technology,
{manos,rperdisc,dagon,wenke,feamster}@cc.gatech.edu

## Abstract

*The Domain Name System (DNS) is an essential protocol used by both legitimate Internet applications and cyber attacks. For example, botnets rely on DNS to support agile command and control infrastructures. An effective way to disrupt these attacks is to place malicious domains on a "blocklist" (or "blacklist") or to add a filtering rule in a firewall or network intrusion detection system. To evade such security countermeasures, attackers have used* DNS agility*, e.g., by using new domains daily to evade static blacklists and firewalls. In this paper we propose Notos, a dynamic reputation system for DNS. The premise of this system is that malicious, agile use of DNS has unique characteristics and can be distinguished from legitimate, professionally provisioned DNS services. Notos uses passive DNS query data and analyzes the network and zone features of domains. It builds models of known legitimate domains and malicious domains, and uses these models to compute a reputation score for a new domain indicative of whether the domain is malicious or legitimate. We have evaluated Notos in a large ISP's network with DNS traffic from 1.4 million users. Our results show that Notos can identify malicious domains with high accuracy (true positive rate of 96.8%) and low false positive rate (0.38%), and can identify these domains weeks or even months before they appear in public blacklists.*

## 1 Introduction

The Domain Name System (DNS) [12, 13] maps domain names to IP addresses, and provides a core service to applications on the Internet. DNS is also used in network security to distribute IP reputation information, e.g., in the form of DNS-based Block Lists (DNSBLs) used to filter spam [18, 5] or block malicious web pages [26, 14].

Internet-scale attacks often use DNS as well because they are essentially Internet-scale malicious applications. For example, *spyware* uses anonymously registered domains to exfiltrate private information to *drop sites*. Disposable domains are used by *adware* to host malicious or false advertising content. *Botnets* make agile use of short-lived domains to evasively move their command-and-control (C&C) infrastructure. Fast-flux networks rapidly change DNS records to evade blacklists and resist take downs [25]. In an attempt to evade domain name blacklisting, attackers now make very aggressive use of *DNS agility*. The most common example of an *agile* malicious resource is a fast-flux network, but DNS agility takes many other forms including disposable domains (e.g., tens of thousands of randomly generated domain names used for spam or botnet C&C), domains with dozens of `A` records or `NS` records (in excess of levels recommended by RFCs, in order to resist takedowns), or domains used for only a few hours of a botnet's lifetime. Perhaps the best example is the Conficker.C *worm* [15]. After Conficker.C infects a machine, it will try to contact its C&C server, chosen at random from a list of 50,000 possible domain names created every day. Clearly, the goal of Conficker.C was to frustrate blacklist maintenance and takedown efforts. Other malware that abuse DNS include Sinowal (a.k.a. Torpig) [9], Kraken [20], and Srizbi [22]. The aggressive use of newly registered domain names is seen in other contexts, such as spam campaigns and malicious flux networks [25, 19]. This strategy delays takedowns, degrades the effectiveness of blacklists, and pollutes the Internet's name space with unwanted, discarded domains.

In this paper, we study the problem of *dynamically* assigning *reputation* scores to new, unknown domains. Our main goal is to automatically assign a low reputation score to a domain that is involved in malicious activities, such as malware spreading, phishing, and spam campaigns. Conversely, we want to assign a high reputation score to domains that are used for legitimate purposes. The reputation scores enable *dynamic* domain name blacklists to counter cyber attacks much more effectively. For example, with static blacklisting, by the time one has sufficient evidence to put a domain on a blacklist, it typically has been involved in malicious activities for a significant period of time. With dynamic blacklisting our goal is to decide, even for a new domain, whether it is likely used for malicious purposes. To this end, we propose Notos, a system that dynamically assigns reputation scores to domain names. Our work is based on the observation that agile malicious uses of DNS have unique characteristics, and can be distinguished from legitimate, professionally provisioned DNS services. In short, network resources used for malicious and

fraudulent activities inevitably have distinct *network characteristics* because of their need to evade security countermeasures. By identifying and measuring these features, Notos can assign appropriate reputation scores.

Notos uses historical DNS information collected passively from multiple recursive DNS resolvers distributed across the Internet to build a model of how network resources are allocated and operated for legitimate, professionally run Internet services. Notos also uses information about malicious domain names and IP addresses obtained from sources such as spam-traps, honeynets, and malware analysis services to build a model of how network resources are typically allocated by Internet miscreants. With these models, Notos can assign reputation scores to new, previously unseen domain names, therefore enabling dynamic blacklisting of unknown malicious domain names and IP addresses.

Previous work on dynamic reputation systems mainly focused on IP reputation [24, 31, 1, 21]. To the best of our knowledge, our system is the first to create a comprehensive *dynamic* reputation system around domain names. To summarize, our main contributions are as follows:

- We designed Notos, a dynamic, comprehensive reputation system for DNS that outputs reputation scores for domains. We constructed *network* and *zone* features that capture the characteristics of resource provisioning, usages, and management of domains. These features enable Notos to learn models of how legitimate and malicious domains are operated, and compute accurate reputation scores for new domains.

- We implemented a proof-of-concept version of our system, and deployed it in a large ISP's DNS network in Atlanta, GA and San Jose, CA, USA, where we observed DNS traffic from 1.4 million users. We also used passive DNS data from Security Information Exchange (SIE) project [3]. This extensive *real-world* evaluation shows Notos can correctly classify new domains with a low false positive rate (0.38%) and high true positive rate (96.8%). Notos can detect and assign a low reputation score to malware- and spam-related domain names several days or even weeks before they appear on public blacklists.

Section 2 provides some background on DNS and related works. Readers familiar with this may skip to Section 3, where we describe our passive DNS collection strategy and other whitelist and blacklist inputs. We also describe three feature extraction modules that measure key network, zone and evidence-based features. Finally, we describe how these features are clustered and incorporated into the final reputation engine. To evaluate the output of Notos, we gathered an extensive amount of network trace data. Section 4 describes the data collection process, and Section 5 details the sensitivity of each module and final output.

## 2 Background and Related Work

DNS is the protocol that resolves a domain name, like `www.example.com`, to its corresponding IP address, for example `192.0.2.10`. To resolve a domain, a host typically needs to consult a local recursive DNS server (RDNS). A recursive server iteratively discovers which Authoritative Name Server (ANS) is responsible for each zone. The typical result of this iterative process is the mapping between the requested domain name and its current IP addresses.

By aggregating all **unique**, successfully resolved `A-type` DNS answers at the recursive level, one can build a passive DNS database. This passive DNS (pDNS) database is effectively the DNS fingerprint of the monitored network and typically contains unique `A-type` resource records (RRs) that were part of monitored DNS answers. A typical RR for the domain name `example.com` has the following format: {`example.com. 78366 IN A 192.0.2.10`}, which lists the domain name, TTL, class, type, and rdata. For simplicity, we will refer to an RR in this paper as just a tuple of the domain name and IP address.

Passive DNS data collection was first proposed by Florian Weimer [27]. His system was among the first that appeared in the DNS community with its primary purpose being the conversion of historic DNS traffic into an easily accessible format. Zdrnja et al. [29] with their work in "Passive Monitoring of DNS Anomalies" discuss how pDNS data can be used for gathering security information from domain names. Although they acknowledge the possibility of creating a DNS reputation system based on passive DNS measurement, they do not quantify a reputation function. Our work uses the idea of building passive DNS information only as a seed for computing statistical DNS properties for each successful DNS resolution. The analysis of these statistical properties is the basic building block for our dynamic domain name reputation function. Plonka et al. [17] introduced Treetop, a scalable way to manage a growing collection of passive DNS data and at the same time correlate zone and network properties. Their cluster zones are based on different classes of networks (class A, class B and class C). Treetop differentiates DNS traffic based on whether it complies with various DNS RFCs and based on the resolution result. Plonka's proposed method, despite being novel and highly efficient, offers limited DNS security information and cannot assign reputation scores to records.

Several papers, e.g., Sinha et al. [24] have studied the effectiveness of IP blacklists. Zhang, et al. [31] showed that the hit rate of highly predictable blacklists (HBLs) decreases significantly over a period of time. Our work addresses the dynamic DNS blacklisting problem that makes it significantly different from the highly predictable blacklists. Importantly, Notos does not aim to create IP blacklists. By using properties of the DNS protocol, Notos can rank a domain name as potentially malicious or not. Garera et al. [8] discussed "phishing" detection predominately using properties of the URL and not statistical observations about the domains or the IP address. The statistical features used by Holz et al. [10] to detect fast flux networks are similar to the ones we used in our work, however, Notos utilizes a more complete collection of network statistical features and is not limited to fast flux networks detection.

Researchers have attempted to use unique characteristics of malicious networks to detect sources of malicious activity. Anderson et al. [1] proposed Spamscatter as the first system to identify and characterize spamming infrastructure by utilizing layer 7 analysis (i.e., web sites and images in spam). Hao et al. [21] proposed SNARE, a spatio-temporal reputation engine for detecting spam messages with very high accuracy and low false positive rates. The SNARE reputation engine is the first work that utilized statistical network-based features to harvest information for spam detection. Notos is complementary to SNARE and Spamscatter, and extends both to not only detect spam, but also identify other malicious activity such as phishing and malware hosting. Qian et al. [28] present their work on spam detection using network-based clustering. In this work, they show that network-based clusters can increase the accuracy of spam-oriented blacklists. Our work is more general, since we try to identify various kinds of malicious domain names. Nevertheless, both works leverage network-based clustering for identifying malicious activities.

Felegyhazi et al. [7] proposed a DNS reputation blacklisting methodology based on `WHOIS` observations. Our system does not use `WHOIS` information making our approaches complementary by design. Sato et al. [23] proposed a way to extend current blacklists by observing the co-occurrence of IP address information. Notos is a more generic approach than the proposed system by Sato and is not limited to botnet related domain name detection. Finally, Notos builds the reputation function mainly based upon passive information from DNS traffic observed in real networks — not traffic observed from honeypots.

No previous work has tried to assign a dynamic domain name reputation score for any domain that traverses the edge of a network. Notos harvests information from multiple sources—the domain name, its effective zone, the IP address, the network the IP address belongs to, the Autonomous System (AS) and honeypot analysis. Furthermore, Notos uses short-lived passive DNS information. Thus, it is difficult for a malicious domain to dilute its passive DNS footprint.

## 3 Notos: A Dynamic Reputation System

The goal of the Notos reputation system is to dynamically assign reputation scores to domain names. Given a domain name $d$, we want to assign a low reputation score if $d$ is involved in malicious activities (e.g., if it has been involved with botnet C&C servers, spam campaigns, malware propagation, etc.). On the other hand, we want to assign a high reputation score if $d$ is associated with legitimate Internet services.

Notos' main source of information is a passive DNS (pDNS) database, which contains historical information about domain names and their resolved IPs. Our pDNS database is constantly updated using real-world DNS traffic from multiple geographically diverse locations as shown in Figure 1. We collect DNS traffic from two ISP recursive DNS servers (RDNS) located in Atlanta and San Jose. The ISP nodes witness 30,000 DNS queries/second during peak hours. We also collect DNS traffic through the Security Information Exchange (SIE) [3], which aggregates DNS traffic received by a large number of RDNS servers from authoritative name servers across North America and Europe. In total, the SIE project processes approximately 200 Mbit/s of DNS messages, several times the total volume of DNS traffic in a single US ISP.

Another source of information we use is a list of known malicious domains. For example, we run known malware samples in a controlled environment and we classify as suspicious all the domains contacted by malware samples that do not match a pre-compiled white list. In addition, we extract suspicious domain names from spam emails collected using a large spam-trap. Again, we discard the domains that match our whitelist and consider the rest as potentially malicious. Furthermore, we collect a large list of popular, legitimate domains from `alexa.com` (we discuss our data collection and analysis in more details in Section 4). The set of known malicious and legitimate domains represents our *knowledge base*, and is used to train our reputation engine, as we discuss in Section 4.

Intuitively, a domain name $d$ can be considered suspicious when there is evidence that $d$ or its IP addresses are (or were in previous months) associated with known malicious activities. The more evidence of "bad associations" we can find about $d$, the lower the reputation score we will assign to it. On the other hand, if there is evidence that $d$ is (or was in the past) associated with legitimate, professionally run Internet services, we will assign it a higher reputation score.

### 3.1 System Overview

Before describing the internals of our reputation system, we introduce some basic terminology. A domain name $d$ consists of a set of substrings or labels separated by a period; the rightmost label is called the *top-level* domain, or TLD. The *second-level* domain (2LD) represents the two rightmost labels separated by a period; the *third-level* domain (3LD) analogously contains the three rightmost labels, and so on. As an example, given the domain name $d$="a.b.example.com", $TLD(d)$="com", $2LD(d)$="example.com", and $3LD(d)$="b.example.com".

Let $s$ be a domain name (e.g., $s$="example.com"). We define $Zone(s)$ as the set of domains that include $s$ and all domain names that end with a period followed by $s$ (e.g., domains ending in ".example.com").

Let $D = \{d_1, d_2, ..., d_m\}$ be a set of domain names. We

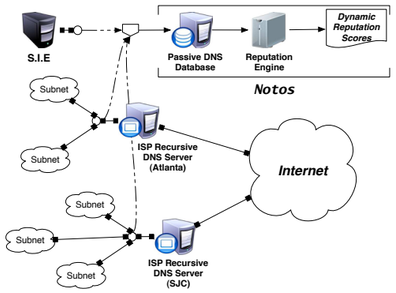



**Figure 1. System overview.**

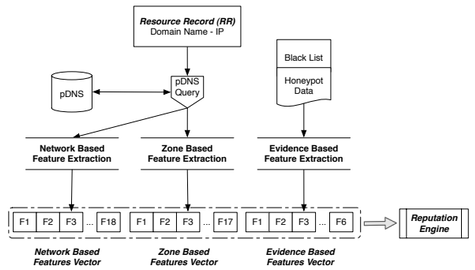



**Figure 2. Computing network-based, zone-based, evidence-based features.**

call $A(D)$ the set of IP addresses ever pointed to by any domain name $d \in D$.

Given an IP address $a$, we define $BGP(a)$ to be the set of all IPs within the BGP prefix of $a$, and $AS(a)$ as the set of IPs located in the autonomous system in which $a$ resides. In addition, we can extend these functions to take as input a set of IPs: given IP set $A = a_1, a_2, ..., a_N$, $BGP(A) = \bigcup_{k=1..N} BGP(a_k)$; $AS(a)$ is similarly extended.

To assign a reputation score to a domain name $d$ we proceed as follows. First, we consider the most current set $A_c(d) = \{a_i\}_{i=1..m}$ of IP addresses to which $d$ points. Then, we query our pDNS database to retrieve the following information:

- *Related Historic IPs (RHIPs)*, which consist of the union of $A(d)$, $A(Zone(3LD(d)))$, and $A(Zone(2LD(d)))$. In order to simplify the notation we will refer to $A(Zone(3LD(d)))$ and $A(Zone(2LD(d)))$ as $A_{3LD}(d)$ and $A_{2LD}(d)$, respectively.

- *Related Historic Domains (RHDNs)*, which comprise the entire set of domain names that ever resolved to an IP address $a \in AS(A(d))$. In other words, RHDNs contain all the domains $d_i$ for which $A(d_i) \cap AS(A(d)) \neq \emptyset$.

After extracting the above information from our pDNS database, we measure a number of statistical features. Specifically, for each domain $d$ we extract three groups of features, as shown in Figure 2:

- **Network-based features:** The first group of statistical features is extracted from the set of RHIPs. We measure quantities such as the total number of IPs historically associated with $d$, the diversity of their geographical location, the number of distinct autonomous systems (ASs) in which they reside, etc.

- **Zone-based features:** The second group of features we extract are those from the RHDNs set. We measure the average length of domain names in RHDNs, the number of distinct TLDs, the occurrence frequency of different characters, etc.

- **Evidence-based features:** The last set of features includes the measurement of quantities such as the number of distinct malware samples that contacted the domain $d$, the number of malware samples that connected to any of the IPs pointed by $d$, etc.

Once extracted, these statistical features are fed to the reputation engine. Notos' reputation engine operates in two modes: an off-line "training" mode and an on-line "classification" mode. During the off-line mode, Notos *trains* the reputation engine using the information gathered in our *knowledge base*, namely the set of known malicious and legitimate domain names and their related IP addresses. Afterwards, during the on-line mode, for each new domain $d$, Notos queries the trained reputation engine to compute a reputation score for $d$ (see Figure 3). We now explain the details about the statistical features we measure, and how the reputation engine uses them during the off-line and on-line modes to compute a domain names' reputation score.

## 3.2 Statistical Features

In this section we identify key statistical features and the intuition behind their selection.

### 3.2.1 Network-based Features

Given a domain $d$ we extract a number of statistical features from the set RHIPs of $d$, as mentioned in Section 3.1. Our network-based features describe how the operators who *own* $d$ and the IPs that domain $d$ points to, allocate their network resources. Internet miscreants often abuse DNS to operate their malicious networks with a high level of *agility*. Namely, the



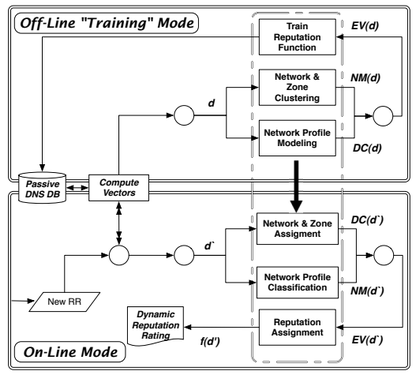



**Figure 3. Off-line and on-line modes in Notos.**

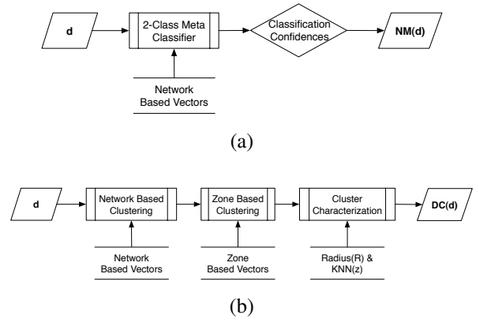



**Figure 4. (a) Network profile modeling in Notos. (b) Network and zone based clustering in Notos.**

domain names and IPs that are used for malicious purposes are often short-lived and are characterized by a high *churn* rate. This agility avoids simple blacklisting or removals by law enforcement. In order to measure the level of agility of a domain name $d$, we extract eighteen statistical features that describe $d$'s *network profile*. Our network features fall into the following three groups:

- *BGP features*. This subset consists of a total of nine features. We measure the number of distinct BGP prefixes related to $BGP(A(d))$, the number of countries in which these BGP prefixes reside, and the number of organizations that *own* these BGP prefixes; the number of distinct IP addresses in the sets $A_{3LD}(d)$ and $A_{2LD}(d)$; the number of distinct BGP prefixes related to $BGP(A_{3LD}(d))$ and $BGP(A_{2LD}(d))$, and the number of countries in which these two sets of prefixes reside.

- *AS features*. This subset consists of three features, namely the number of distinct autonomous systems related to $AS(A(d))$, $AS(A_{3LD}(d))$, and $AS(A_{2LD}(d))$.

- *Registration features*. This subset consists of six features. We measure the number of distinct registrars associated with the IPs in the $A(d)$ set; the diversity in the registration dates related to the IPs in $A(d)$; the number of distinct registrars associated with the IPs in the $A_{3LD}(d)$ and $A_{2LD}(d)$ sets; and the diversity in the registration dates for the IPs in $A_{3LD}(d)$ and $A_{2LD}(d)$.

While most legitimate, professionally run Internet services have a very stable network *profile*, which is reflected into low values of the network features described above, the profiles of malicious networks (e.g., fast-flux networks) usually change relatively frequently, thus causing their network features to be assigned higher values. We expect a domain name $d$ from a legitimate zone to exhibit a small values in its AS features, mainly because the IPs in the RHIPs should belong to the same organization or a small number of different organizations. On the other hand, if a domain name $d$ participates in malicious activities (i.e., botnet activities, flux networks), then it could reside in a large number of different networks. The list of IPs in the RHIPs that correspond to the malicious domain name will produce AS features with higher values. In the same sense, we measure that homogeneity of the registration information for benign domains. Legitimate domains are typically linked to address space owned by organizations that acquire and announce network blocks in some order. This means that the registration-feature values for a legitimate domain name $d$ that owned by the same organizations will produce a list of IPs in the RHIPs that will have small registration feature values. If this set of IPs exhibits high registration feature values, it means that they very likely reside in different registrars and were registered on different dates. Such registration-feature properties are typically linked with fraudulent domains.

### 3.2.2 Zone-based Features

The network-based features measure a number of characteristics of IP addresses historically related to a given domain name $d$. On the other hand, the zone-based features measure the characteristics of domain names historically associated with $d$. The intuition behind the zone-based features is that while legitimate Internet services may be associated with many different domain names, these domain names usually have strong similarities. For example, `google.com`, `googlesyndication.com`, `googlewave.com`, etc., are all related to Internet services provided by Google, and contain the string "google" in their name. On the other hand, malicious domain names related to the same spam campaign, for example, often look randomly generated and share few common characteristics. Therefore, our zone-based features aim to measure the

level of *diversity* across the domain names in the RHDNs set. Given a domain name $d$, we extract seventeen statistical features that describe the properties of the set RHDNs of domain names related to $d$. We divide these seventeen features into two groups:

- *String features*. This group consists of twelve features. We measure the number of distinct domain names in RHDNs, and the average and standard deviation of their length; the mean, median, and standard deviation of the occurrence frequency of each single character in the domain name strings in RHDNs; the mean, median and standard deviation of the distribution of 2-grams (i.e., pairs of characters); the mean, median and standard deviation of the distribution of 3-grams.

- *TLD features*. This group consists of five features. For each domain $d_i$ in the RHDNs set, we extract its top-level domain $TLD(d_i)$ and we count the number of distinct TLD strings that we obtain; we measure the ratio between the number of domains $d_i$ whose $TLD(d_i)$=".com" and the total number of TLD different from ".com"; also, we measure the mean, median, and standard deviation of the occurrence frequency of the TLD strings.

It is worth noting that whenever we measure the mean, median and standard deviation of a certain property, we do so in order to summarize the shape of its distribution. For example, by measuring the mean, median, and standard deviation of the occurrence frequency of each character in a set of domain name strings, we summarize how the distribution of the character frequency looks like.

#### 3.2.3 Evidence-based Features

We use the evidence-based features to determine to what extent a given domain $d$ is associated with other known malicious domain names or IP addresses. As mentioned above, Notos collects a *knowledge base* of known suspicious, malicious, and legitimate domain names and IPs from public sources. For example, we collect malware-related domain names by executing large numbers of malware samples in a controlled environment. Also, we check IP addresses against a number of public IP blacklists. We elaborate on how we build Notos' knowledge base in Section 4. Given a domain name $d$, we measure six statistical features using the information in the knowledge base. We divide these features into two groups:

- *Honeypot features*. We measure three features, namely the number of distinct malware samples that, when executed, try to contact $d$ or any IP address in $A(d)$; the number of malware samples that contact any IP address in $BGP(A(d))$; and the number of samples that contact any IP address in $AS(A(d))$.

- *Blacklist features*. We measure three features, namely the number of IP addresses in $A(d)$ that are listed in public IP blacklists; the number of IPs in $BGP(A(d))$ that are listed in IP blacklists; and the number of IPs in $AS(A(d))$ that are listed in IP blacklists.

Notos uses the blacklist features from the evidence vector so it can identify the re-use of known malicious network resources like IPs, BGP prefixes or even ASs. Domain names are significantly cheaper than IPv4 addresses; so malicious users tend to reuse address space with new domain names. We should note that the evidence-based features represent only part of the information we used to compute the reputation scores. The fact that a domain name was queried by malware does not automatically mean that the domain will receive a low reputation score.

### 3.3 Reputation Engine

Notos' reputation engine is responsible for deciding whether a domain name $d$ has characteristics that are similar to either legitimate or malicious domain names. In order to achieve this goal, we first need to *train* the engine to recognize whether $d$ belongs (or is "close") to a known *class of domains*. This training can be repeated periodically, in an off-line fashion, using historical information collected in Notos' *knowledge base* (see Section 4). Once the engine has been trained, it can be used in on-line mode to assign a reputation score to each new domain name $d$.

In this section, we first explain how the reputation engine is trained, and then we explain how a trained engine is used to assign reputation scores.

#### 3.3.1 Off-Line Training Mode

During off-line training (Figure 3), the reputation engine builds three different modules. We briefly introduce each module and then elaborate on the details.

- *Network Profiles Model*: a model of how well known networks behave. For example, we model the network characteristics of popular content delivery networks (e.g., Akamai, Amazon CloudFront), and large *popular* websites (e.g., google.com, yahoo.com). During the on-line mode, we compare each new domain name $d$ to these models of well-known network profiles, and use this information to compute the final reputation score, as explained below.

- *Domain Name Clusters*: we group domain names into clusters sharing similar characteristics. We create these clusters of domains to identify groups of domains that contain mostly malicious domains, and groups that contain mostly legitimate domains. In the on-line mode, given a new domain $d$, if $d$ (more precisely, $d$'s projection into a statistical feature space) falls within (or close to) a cluster of domains containing mostly malicious domains, for example, this gives us a hint that $d$ should be assigned a low reputation score.

- *Reputation Function*: for each domain name $d_i, i = 1..n$, in Notos' knowledge base, we *test* it against the trained network profiles model and domain name clusters. Let $NM(d_i)$ and $DC(d_i)$ be the output of the Network Profiles (NP) module and the Domain Clusters (DC) module, respectively. The reputation function takes in input $NM(d_i)$, $DC(d_i)$, and information about whether $d_i$ and its resolved IPs $A(d_i)$ are known to be legitimate, suspicious, or malicious (i.e., if they appeared in a domain name or IP blacklist), and builds a model that can assign a reputation score between zero and one to $d$. A reputation score close to zero signifies that $d$ is a malicious domain name while a score close to one signifies that $d$ is benign.

We now describe each module in detail.

#### 3.3.2 Modeling Network Profiles

During the off-line training mode, the reputation engine builds a model of well-known network behaviors. An overview of the network profile modeling module can be seen in Figure 4(a). In practice we select five sets of domain names that share similar characteristics, and *learn* their network profiles. For example, we identify a set of domain names related to very popular websites (e.g., google.com, yahoo.com, amazon.com) and for each of the related domain names we extract their network features, as explained in Section 3.2.1. We then use the extracted feature vectors to train a statistical classifier that will be able to recognize whether a new domain name $d$ has network characteristics similar to the popular websites we modeled.

In our current implementation of Notos we model the following classes of domain names:

- **Popular Domains**. This class consists of a large set of domain names under the following DNS zones: google.com, yahoo.com, amazon.com, ebay.com, msn.com, live.com, myspace.com, and facebook.com.

- **Common Domains**. This class of domains includes domain names under the top one hundred zones, according to `alexa.com`. We exclude from this group all the domain names already included in the *Popular Domains* class (which we model separately).

- **Akamai Domains**. Akamai is a large content delivery network (CDN), and the domain names related to this CDN have very peculiar network characteristics. To model the network profile of Akamai's domain names, we collect a set of domains under the following zones: akafms.net, akamai.net, akamaiedge.net, akamai.com, akadns.net, and akamai.com.

- **CDN Domains**. In this class we include domain names related to CDNs other than Akamai. For example, we collect domain names under the following zones: panthercdn.com, llnwd.net, cloudfront.net, nyud.net, nyucd.net and redcondor.net. We chose not to aggregate these CDN domains and Akamai's domains in one class, since we observed that Akamai's domains have a very unique network profile, as we discuss in Section 4. Therefore, learning two separate models for the classes of *Akamai Domains* and *CDN Domains* allows use to achieve better classification accuracy during the on-line mode, compared to learning only one model for both classes (see Section 3.3.5).

- **Dynamic DNS Domains**. This class includes a large set of domain names registered under two of the largest dynamic DNS providers, namely No-IP (`no-ip.com`) and DynDNS (`dyndns.com`).

For each class of domains, we train a statistical classifier to distinguish between one of the classes and all the others. Therefore, we train five different classifiers. For example, we train a classifier that can distinguish between the class of *Popular Domains* and all other classes of domains. That is, given a new domain name $d$, this classifier is able to recognize whether $d$'s network profile looks like the profile of a well-known popular domain or not. Following the same logic we, can recognize network profiles for the other classes of domains.

#### 3.3.3 Building Domain Name Clusters

In this phase, the reputation engine takes the domain names collected in our pDNS database during a *training period*, and builds clusters of domains that share similar network and zone based features. The overview of this module can be seen in Figure 4(b). We perform clustering in two steps. In the first step we only use the network-based features to create coarse-grained clusters. Then, in the second step, we split each coarse-grained cluster into finer clusters using only the zone-based features, as shown in Figure 5.

**Network-based Clustering** The objective of network-based clustering is to group domains that share similar levels of *agility*. This creates separate clusters of domains with "stable" network characteristics and "non-stable" networks (like CDNs and malicious flux networks).

**Zone-based Clustering** After clustering the domain names according to their network-based features, we further split the network-based clusters of domain names into finer groups. In this step, we group domain names that are in the same

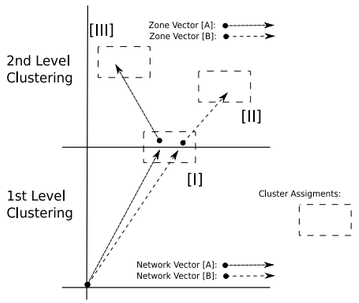



**Figure 5. Network & zone based clustering process in Notos, in the case of a Akamai [A] and a malicious [B] domain name.**

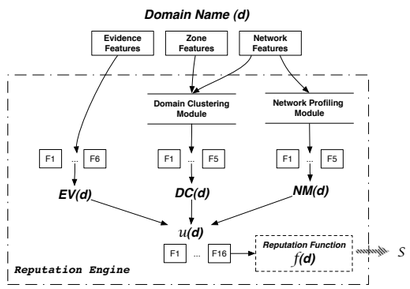



**Figure 6. The output from the network profiling module, the domain clustering module and the evidence vector will assist the reputation function to assign the reputation score to the domain $d$.**

network-based cluster and also share similar zone-based features. To better understand how the zone-based clustering works, consider the following examples of zone-based clusters:

**Cluster 1**:

..., **72.247.176.81 e55.g.akamaiedge.net,** 72.247.176.94 e68.g.akamaiedge.net, 72.247.176.146 e120.g.akamaiedge.net, 72.247.176.65 e39.na.akamaiedge.net, 72.247.176.242 e216.g.akamaiedge.net, 72.247.176.33 e7.g.akamaiedge.net, 72.247.176.156 e130.g.akamaiedge.net, 72.247.176.208 e182.g.akamaiedge.net, 72.247.176.198 e172.g.akamaiedge.net, 72.247.176.217 e191.g.akamaiedge.net, 72.247.176.200 e174.g.akamaiedge.net, 72.247.176.99 e73.g.akamaiedge.net, 72.247.176.103 e77.g.akamaiedge.net, 72.247.176.59 e33.c.akamaiedge.net, 72.247.176.68 e42.gb.akamaiedge.net, 72.247.176.237 e211.g.akamaiedge.net, 72.247.176.71 e45.g.akamaiedge.net, 72.247.176.239 e213.na.akamaiedge.net, 72.247.176.120 e94.g.akamaiedge.net, ...

**Cluster 2**:

..., **90.156.145.198 spzr.in,** 90.156.145.198 vwui.in, 90.156.145.198 x9e.ru, 90.156.145.50 v2802.vps.masterhost.ru, 90.156.145.167 www.inshaker.ru, 90.156.145.198 x71.ru, 90.156.145.198 c3q.at, 90.156.145.198 ltkq.in, 90.156.145.198 x7d.ru, 90.156.145.198 zdlz.in, 90.156.145.159 www.designcollector.ru, 90.156.145.198 x7o.ru, 90.156.145.198 q5c.ru, 90.156.145.159 designtwitters.com, 90.156.145.198 u5d.ru, 90.156.145.198 x9d.ru, 90.156.145.198 xb8.ru, 90.156.145.198 xg8.ru, 90.156.145.198 x8m.ru, 90.156.145.198 shopfilmworld.cn, 90.156.145.198 bigappletopworld.cn, 90.156.145.198 uppd.in, ...

Each element of the cluster is a *domain name - IP address* pair. These two groups of domains belonged to the same network cluster, but were separated into two different clusters by the zone-based clustering phase. *Cluster 1* contains domain names belonging to Akamai's CDN, while the domains in *Cluster 2* are all related to malicious websites that distribute malicious software. The two clusters of domains share similar network characteristics, but have significantly different zone-based features. For example, consider domain names $d_1$="e55.g.akamaiedge.net" from the first cluster, and $d_2$="spzr.in" from the second cluster. The reason why $d_1$ and $d_2$ were clustered in the same network-based cluster is because the set of RHIPs (see Section 3.1) for $d_1$ and $d_2$ have similar characteristics. In particular, the *network agility* properties of $d_2$ make it look like if it was part of a large CDN. However, when we consider the set of RHDNs for $d_1$ and $d_2$, we can notice that the zone-based features of $d_1$ are much more "stable" than the zone-based features of $d_2$. In other words, while the RHDNs of $d_1$ share strong domain name similarities (e.g., they all share the substring "akamai") and have low variance of the *string features* (see Section 3.2.2), the strong *zone agility* properties of $d_2$ affect the zone-based features measured on $d_2$'s RHDNs and make $d_2$ look very different from $d_1$.

One of the main advantages of Notos is the reliable assignment of low reputation scores to domain names participating in "agile" malicious campaigns. Less agile malicious campaigns, e.g., Fake AVs campaigns may use domain names structured to resemble CDN related domains. Such strategies would not be beneficial for the FakeAV campaign, since domains like `virus-scan1.com`, `virus-scan2.com`, etc., can be trivially blocked by using simple regular expressions [16]. In other words, the attackers need to introduce more "agility" at both the network and domain name level in order to avoid simple domain name blacklisting. Notos would only require a few labeled domain names belonging to the malicious campaign for training purposes, and the reputation engine would then generalize to assign a low reputation score to the remaining (previously unknown) domain names that belong to the same malicious campaign.

#### 3.3.4 Building the Reputation Function

Once we build a model of well-known network profiles (see Section 3.3.2) and the domain clusters (see Section 3.3.3), we can build the reputation function. The reputation function will assign a reputation score in the interval $[0, 1]$ to domain names, with 0 meaning low reputation (i.e., likely malicious) and 1 meaning high reputation (i.e., likely legitimate). We implement our reputation function as a statistical classifier. In order to train the reputation function, we consider all the domain names $d_i, i = 1, .., n$ in Notos' *knowledge base*, and we feed each domain $d_i$ to the *network profiles* module and to the *domain clusters* module to compute two output vectors $NM(d_i)$ and $DC(d_i)$, respectively. We explain the details of how $NM(d_i)$ and $DC(d_i)$ are computed later in Section 3.3.5. For now it sufficient to consider $NM(d_i)$ and $DC(d_i)$ as two feature vectors. For each $d_i$ we also compute an *evidence features* vector $EV(d_i)$, as described in Section 3.2.3. Let $v(d_i)$ be a feature vector that combines the $NM(d_i)$, $DC(d_i)$, and $EV(d_i)$ feature vectors. We train the reputation function using the labeled dataset $L = \{(v(d_i), y_i)\}_{i=1..n}$, where $y_i = 0$ if $d_i$ is a known malicious domain name, otherwise $y_i = 1$.

#### 3.3.5 On-Line Mode

After training is complete; the reputation engine can be used in on-line mode (Figure 3) to assign a reputation score to new domain names. For example, given an input domain name $d$, the reputation engine computes a score $S \in [0, 1]$. Values of $S$ close to zero mean that $d$ appears to be related to malicious activities and therefore has a low reputation. On the other hand, values of $S$ close to one signify that $d$ appears to be associated with benign Internet services, and therefore has a high reputation. The reputation score is computed as follows. First, $d$ is fed into the *network profiles* module, which consists of five statistical classifiers, as discussed in Section 3.3.2. The output of the *network profiles* module is a vector $NM(d) = \{c_1, c_2, ..., c_5\}$, where $c_1$ is the output of the first classifier, and can be viewed as the probability that $d$ belongs to the class of *Popular Domains*, $c_2$ is the probability that $d$ belongs to the class of *Common Domains*, etc. At the same time, $d$ is fed into the *domain clusters* module, which computes a vector $DC(d) = \{l_1, l_2, ..., l_5\}$. The elements $l_i$ of this vector are computed as follows. Given $d$, we first extract its network-based features and identify the closest network-based cluster to $d$, among the network-based clusters computed by the *domain clusters* module during the off-line mode (see Section 3.3.3). Then, we extract the zone-based statistical features and identify the zone-based cluster closest to $d$. Let this closest domain cluster be $C_d$. At this point, we consider all the zone-based feature vectors $v_j \in C_d$, and we select the subset of vectors $V_d \subseteq C_d$ for which the two following conditions are verified: i) $dist(z_d, v_j) < R$, where $z_d$ is the zone-based feature vector for $d$, and $R$ is a predefined *radius*; ii) $v_j \in KNN(z_d)$, where $KNN(z_d)$ is the set of $k$ nearest-neighbors of $z_d$.

The feature vectors in $V_d$ are related to domain names extracted from Notos' *knowledge base*. Therefore, we can assign a label to each vector $v_i \in V_d$, according to the nature of the domain name $d$ from which $v_i$ was computed. The domains in Notos' *knowledge base* belong to different classes. In particular, we distinguish between eight different classes of domains, namely *Popular Domains*, *Common Domains*, *Akamai*, *CDN*, and *Dynamic DNS*, which have the same meaning as explained in Section 3.3.2, and *Spam Domains*, *Flux Domains*, and *Malware Domains*.

In order to compute the output vector $DC(d)$, we compute the following five statistical features: the *majority class* label $L$ (e.g., $L$ may be equal to *Malware Domain*), i.e., the label that appears the most among the vectors $v_i \in V_d$; the standard deviation of label frequencies, i.e., given the occurrence frequency of each label among the vectors $v_i \in V_d$ we compute their standard deviation; given the subset $V_d^{(L)} \subseteq V_d$ of vectors in $V_d$ that are associated with label $L$, we compute the *mean*, *median* and *standard deviation* of the distribution of distances between $z_d$ and the vectors $v_j \in V_d^{(L)}$.

#### 3.3.6 Assigning Reputation Scores

Given a domain $d$, once we compute the vectors $NM(d)$ and $DC(d_i)$ as explained above, we also compute the evidence vector $EV(d)$ as explained in Section 3.2.3. At this point, we concatenate these three feature vectors into a sixteen dimensional feature vector $v(d)$, and we feed $v(d)$ in input to our *trained* reputation function (see Section 3.3.4). The reputation function computes a score $S = 1 - f(d)$, where $f(d)$ can be interpreted as the probability that $d$ is a malicious domain name. $S$ varies in the $[0, 1]$ interval, and the lower the value of $S$, the lower $d$'s reputation.

## 4 Data Collection and Analysis

This section summarizes observations from passive DNS measurements, and how professional, legitimate DNS services are distinguished from malicious services. These observations provided the ground truth for our dynamic domain name reputation system. We also provide an intuitive example to illustrate these properties, using a few major Internet zones like Akamai and Google.

### 4.1 Data Collection

The basic building block for our dynamic reputation rating system is the historical or "passive" information from successful `A-type` DNS resolutions. We use the DNS traffic from two ISP-based sensors, one located on the US east coast (Atlanta) and one located on the US west coast (San Jose). Additionally we use the aggregated DNS traffic from the different networks covered by the SIE [3]. In total, our database collected 27,377,461 unique resolutions from all these sources over a period of 68 days, from $19^{th}$ of July 2009 to $24^{th}$ September 2009.

Simple measurements performed on this large data set demonstrate a few important properties leveraged by our selected features. After just a few days the rate of new, unique pDNS entries leveled off. The graph in Figure 7(b) shows only about 100,000 to 150,000 new domains/day (with a brief outage issue on the $53^{rd}$ day), despite very large numbers of

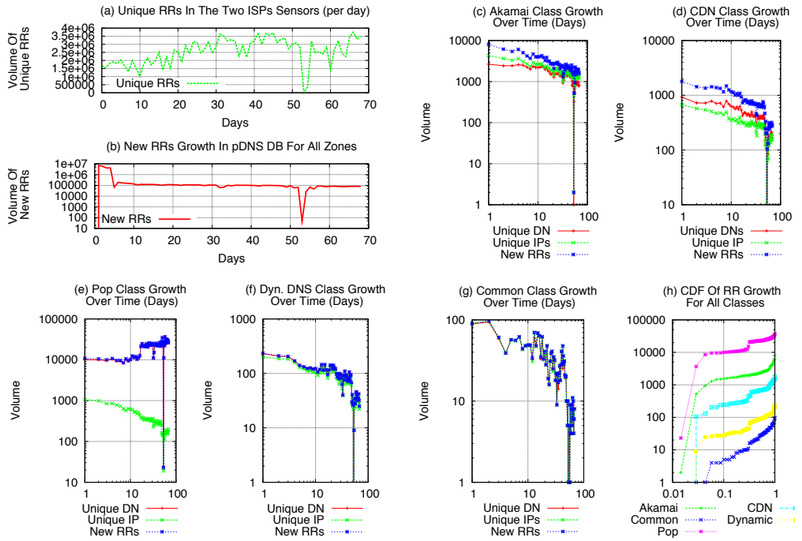



**Figure 7. Various RRs growth trends observed in the pDNS DB over a period of 68 days**

RRs arriving each day (shown in Figure 7(a)). This suggests that most RRs are duplicates, and approximately after the first few days, 94.7% – on average – from the unique RRs observed in daily base at the sensor level are already recorded by the passive DNS database. Therefore, even a relatively small pDNS database may be used to deploy Notos. In Section 5, we measure the sensitivity of our system to traffic collected from smaller networks.

The remaining plots in Figure 7 show the daily growth of our passive DNS database, from the point of view of five different zone classes. Figure 7(c) and (d) show the growth rate associated with CDN networks (Akamai, and all other CDNs). The number of unique IPs stays nearly constant with the number of unique domains (meaning that each new RR is a new IP and a new child domain of the CDN). In a few weeks, most of the IPs became known—suggesting that one can fully map CDNs in a modest training set. This is because CDNs, although large, always have a fixed number of IP addresses used for hosting their high-availability services. Intuitively, we believe this would not be the case with malicious CDNs (e.g., flux networks), which use randomly spreading infections to continually recruit new IPs.

The ratio of new IPs to domains diverges in Figure 7(e), a plot of the rate of newly discovered RRs for popular websites (e.g., Google, Facebook). Facebook notably uses unique child domains for their Web-based chat client, and other top Internet sites use similar strategies (encoding information in the domain, instead of the URI), which explains the growth in domains shown in Figure 7(e). These popular sites use a very small number of IPs, however, and after a few weeks of training our pDNS database identified all of them. Since these popular domains make up a large portion of traffic in any trace, our intuition is that simple whitelisting would significantly reduce the workload of a classifier.

Figure 7(f) shows the rate of pDNS growth for zones in Dynamic DNS providers. These services, sometimes used by botmasters, demonstrate a nearly matched ratio of new IPs to new domains. The data excludes non-routable answers (e.g., dynamic DNS domains pointing to 127.0.0.1), since this contains no unique network information. Intuitively, one can think of dynamic DNS as a nearly complete bijection of domains to IPs. Figure 7(g) shows the growth of RRs for `alexa.com` top 100 domains. Unlike dynamic DNS domains, these points to a small set of unique addresses, and most can be identified in a few weeks' worth of training.

A comparison of all the zone classes appears in Figure 7(h), which shows the cumulative distribution of the unique RRs detailed in Figure 7(c) through (g). The different rates of change illustrate how each zone class has a distinct pattern of RR use: some have a small IP space and highly variable domain names; some pair nearly every new domain with a new IP. Learning approximately 90% of all the unique RRs in each zone class, however, only requires (at most) tens of thousands of distinct RRs. The intuition from this plot is that, despite the very large



data set we used in our study, Notos could potentially work with data observed from much smaller networks.

### 4.2 Building The Ground Truth

To establish ground truth, we use two different labeling processes. First, we assigned labels to RRs at the time of their discovery. This provided an initial static label for many domains. Blacklists, of course, are never complete and always dynamic. So our second labeling process took place during evaluation, and monitored several well-known domain blacklists and whitelists.

The data we used for labeling came from several sources. Our primary source of blacklisting came from services such as `malwaredomainlist.com` and `malwaredomains.com`. In order to label IP addresses in our pDNS database we also used the Sender Policy Block (SBL) list from Spamhaus [18]. Such IPs are either known to send spam or distribute malware. We also collected domain name and IP blacklisting information from the Zeus tracker [30]. All this blacklisting information was gathered before the first day of August 2009 (during all the 15 days in which we collected passive DNS data). Since blacklists traditionally lag behind the active threat, we continued to collect all new data until the end of our experiments.

Our limited whitelisting was derived from the top 500-`alexa.com` domain names, as of the $1^{st}$ of August 2009. We reasoned that, although some malicious domains become popular, they do not stay popular (because of remediation), and never break into the top tier of domain rankings. Likewise, we used a list of the 18 most common 2LDs from various CDNs, which composed the main corpus of our CDN labeled RRs. Finally a list of 464 dynamic DNS second level domains allowed us to identify and label domain name and IPs coming from zones under dynamic DNS providers. We label our evaluation (or testing) data-set by aggregating updated blacklist information for new malicious domain names and IPs from the same lists.

To compute the honeypot features (presented in Section 3.2.3) we need a malware analysis infrastructure that can process as many "new" malware samples as possible. Our honeypot infrastructure is similar to "Ether" [4] and is capable of processing malware samples in a queue. Every malware sample was analyzed in a controlled environment for a time period of five minutes. This process was repeated during the last 15 days of July 2009. After 15 days of executions we obtained a set of successful DNS resolutions (domain names and IPs) that each malware looked up. We chose to execute malware and collect DNS evidence through the same period of time in which we aggregate the passive DNS database. Our virtual machines are equipped with five popular commercial anti-virus engines. If one of the engines identifies an executable as malicious, we capture all domain names and the corresponding IP mappings that the malware used during execution. After excluding all domain names that belong to the top 500 most popular `alexa.com` zones, we assemble the main corpus of our "honeypot data". We automated the crawling and collection of black list information and honeypot execution.

The reader should note that we chose to label our data in as transparent way as possible. We used public blacklisting information to label our training dataset before we build our models and train the reputation function. Then we assigned the reputation scores and validated the results again using the same publicly available blacklist sources. It is safe to assume that private IP and DNS blacklist will contain significant more complete information with lower FP rates than the public blacklists. By using such type of private blacklist the accuracy of Notos' reputation function should improve significantly.

## 5 Results

In this section, we present the experimental results of our evaluation. We show that Notos can identify malicious domain names sooner than public blacklists, with a low false positive rate (FP%) of 0.38% and high true positive rate (TP%) of 96.8%. As a first step, we computed vectors based on the statistical features (described in Section 3.2) from 250,000 unique RRs. This volume corresponds to the average volume of new – previously unseen – RRs observed at two recursive DNS servers in a major ISP in one day, as noted in Section 4, Figure 7(b). These vectors were computed based on historic passive DNS information from the last two weeks of DNS traffic observed on the same two ISP recursive resolvers in Atlanta and San Jose.

### 5.1 Accuracy of Network Profile Modeling

The accuracy of the Meta-Classification system (Figure 4(a)) in the network profile module is critical for the overall performance of Notos. This is because, in the on-line mode, Notos will receive unlabeled vectors which must be classified and correlated with what is already present in our knowledge base. For example, if the classifier receives a new RR and assigns to it the label Akamai with very high confidence, that implies the RR which produced this vector will be part of a network similar to Akamai. However, this does not necessarily mean that it is part of the actual Akamai CDN. We will see in the next section how we can draw conclusions based on the proximity between labeled and unlabeled RRs within the same zone-based clusters. Furthermore, we discuss the accuracy of the Meta-Classifier when modeling each different network profile class (profile classes are described in Section 3.3.2).

Our Meta-Classifier consists of five different classifiers, one for each different class of domains we model. We chose to use a Meta-Classification system instead of a traditional single classification approach because Meta-Classification systems typically perform better than a single statistical classi-

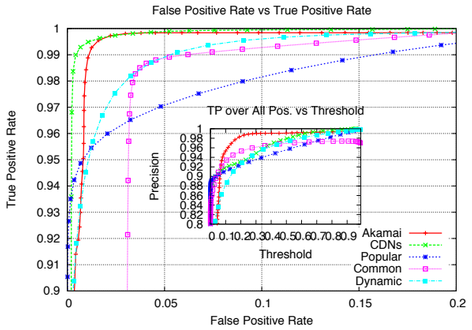



**Figure 8. ROC curves for all network profile classes shows the Meta-Classifier's accuracy.**

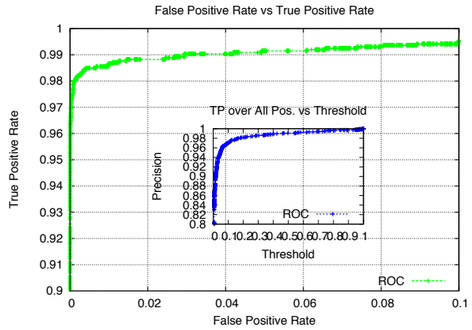



**Figure 9. The ROC curve from the reputation function indicating the high accuracy of Notos.**

fier [11, 2]. Throughout our experiments this proved to be also true. The ROC curve in Figure 8, shows that the Meta-Classifier can accurately classify RRs for all different network profile classes.

The training dataset for the Meta-Classifier is composed of sets of 2,000 vectors from each of the five network profile classes. The evaluation dataset is composed of 10,000 vectors, 2,000 from each of the five network profile classes. The classification results for the domains in the Akamai, CDN, dynamic DNS and Popular classes showed that the supervised learning process in Notos is accurate, with the exception of a small number of false positives related to the Common class (3.8%). After manually analyzing these false positives, we concluded that some level of confusion between the vectors produced by Dynamic DNS domain names and the vectors produced by domain names in the Common class still remains. However, this minor misclassification between network profiles does not significantly affect the reputation function. This is because the zone profiles of the Common and Dynamic DNS domain names are significantly different. This difference in the zone profiles will drive the network-based and zone-based clustering steps to group the RRs from Dynamic DNS class and Common class in different zone-based clusters.

Despite the fact that the network profile modeling process provides accurate results, it doesn't mean this step can independently designate a domain as benign or malicious. The clustering steps will assist Notos to group vectors not only based their network profiles but also based on their zone properties. In the following section we show how the network and zone profile clustering modules can better associate similar vectors, due to properties of their domain name structure.

### 5.2 Network and Zone-Based Clustering Results

In the domain name clustering process (Section 3.3.3, Figure 4(b)) we used X-Means clustering in series, once for the network-based clustering and again for the zone-based clustering. In both steps we set the minimum and maximum number of clusters to one and the total number of vectors in our dataset, respectively. We run these two steps using different numbers of zone and network vectors. Figure 11 shows that after the first 100,000 vectors are used, the number of network and zone clusters remains fairly stable. This means that by computing at least 100,000 network and zone vectors—using a 15-day old passive DNS database—we can obtain a stable population of zone and network based clusters for the monitored network. We should note that reaching this network and cluster equilibrium does not imply that we do not expect to see any new type of domain names in the ISP's DNS recursive. This just denotes that based on the RRs present in our passive DNS database, and the daily traffic at the ISP's recursive, 100,000 vectors are enough to reflect the major network profile trends in the monitored networks. Figure 11 indicates that a sample set of 100,000 vectors may represent the major trends in a DNS sensor. It is hard to safely estimate the exact minimum number of unique RRs that is sufficient to identify all major DNS trends. An answer to this should be based upon the type, size and utilization of the monitored network. Without data from smaller corporate networks it is difficult for us to make a safe assessment about the minimum number of RR necessary for reliably training Notos.

The evaluation dataset we used consisted of 250,000 unique domain names and IPs. The cluster overview is shown in Figure 10 and in the following paragraphs we discuss some interesting observations that can be made from these network-based and zone-based cluster assignments. As an example, network clusters 0 and 1 are predominantly composed of zones participating in fraudulent activities like spam campaigns (yellow) and malware dropping or C&C zones (red). On the other hand, network clusters 2 to 5 contain Akamai, dynamic DNS, and popular zones like Google, all labeled as benign (green). We included the unlabeled vectors (blue) based on which we evaluated the accuracy of our reputation function. We have a sample of unlabeled vectors in almost all network and zone clusters. We will see how already labeled vectors will assist us to characterize the unlabeled vectors in close proximity.

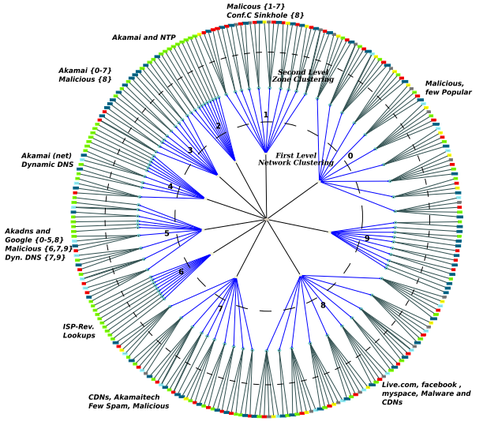



**Figure 10. With the 2-step clustering step, Notos is able to cluster large trends of DNS behavior.**

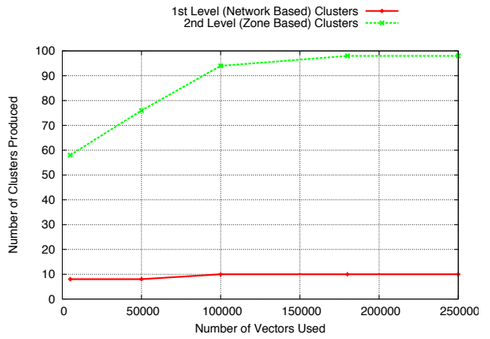



**Figure 11. By using different number of network and zone vectors we observe that after the first 100,000, there is no significant variation in the absolute number of produced clusters during the $1^{st}$ and $2^{nd}$ level clustering steps.**

Before we describe two sample cases of dynamic characterization within zone-based clusters, we need to discuss our radius $R$ and $k$ value selection (see Section 3.3.5). In Section 3.3.5, we discuss how we build domain name clusters. At that point we introduced the dynamic characterization process that gives Notos the ability to utilize already label vectors in order to characterize a newly obtained unlabeled vector by leveraging our prior knowledge. After looking into the distribution of Euclidean distances between unlabeled and labeled vectors within the same zone clusters, we concluded that in the majority of these cases the distances were between 0 and 1000. We tested different values of the radius $R$ and the value of $k$ for the K-nearest neighbors (KNN) algorithm. We observed that the experiments with radius values between 50 and 200 provided the most accurate reputation rating results, which we describe in the following sections. We also observed that if $k > 25$ the accuracy of the reputation function is not affected for all radius values between 50 and 200. Based on the results of these pilot experiments, we decided to set $k$ equal to 50 and the radius distance equal to 100.

Figures 12 and 13 show the effect of this radius selection on two different types of clustering problems. In Figure 12, unknown RRs for `akamaitech.net` are clustered with a labeled vector `akamai.net`. As noted in Section 4, CDNs such as Akamai tended to have new domain names with each RR, but to also reuse their IPs. By training with only a small set of labeled `akamai.net` RRs, our classifier put the new, unknown RRs for `akamaitech.net` into the existing Akamai class. IP-specific features therefore brought the new RRs close to the existing labeled class. Figure 12 compresses all of the dimensions into a two-dimensional plot (for easier visual representation), but it is clear the unknown RRs were all within a distance of 100 to the labeled set.

This result validates the design used in Section 4, where just a few weeks' worth of labeled data was necessary for training. Thus, one does not have to exhaustively discover all whitelisted domains. Notos is resilient to changes in the zone classes we selected. Services like CDNs and major web sites can add new IPs or adjust domain formats, and these will be *automatically* associated with a known labeled class.

The ability of Notos to associate new RRs based on limited labeled inputs is demonstrated again in Figure 13. In this case, labeled Zeus domains (approximately 2,900 RRs from three different Zeus-related BLs) were used to classify new RRs. Figure 13 plots the distance between the labeled Zeus-related RRs and new (previously unknown) RRs that are also related Zeus botnets. As we can see from Section 4, most of the new (unlabeled) Zeus RRs lay very

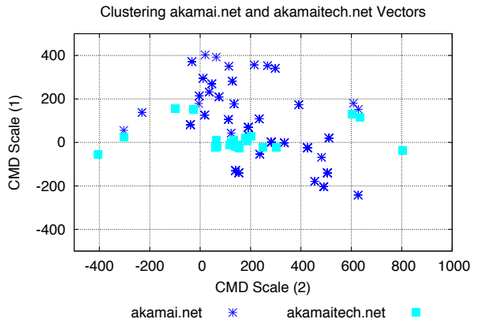



**Figure 12. An example of characterizing the akamaitech.net unknown vectors as benign based on the already labeled vectors (akamai.net) present in the same cluster.**

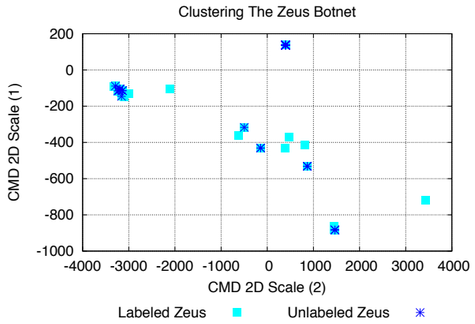



**Figure 13. An example of how the Zeus botnet clusters during our experiments. All vectors are in the same network cluster and in two different zone clusters.**

close, and often even overlap, to known Zeus RRs. This is a good result, because Zeus botnets are notoriously hard to track, given the botnet's extreme agility. Tracking systems such as `zeustracker.abuse.ch` and `malware-domainlist.com` have limited visibility into the botnet, and often produce disjoint blacklists. Notos addresses this problem, by leveraging a limited amount of training data to correctly classify new RRs. During our evaluation set, Notos correctly detected 685 new (previously unknown) Zeus RRs.

### 5.3 Accuracy of the Reputation Function

The first thing that we address in this section is our decision to use a Decision Tree using Logit-Boost strategy (LAD) as the reputation function. Our decision is motivated by the time complexity, the detection results and the precision (true positives over all positives) of the classifier. We compared the LAD classifier to several other statistical classifiers using a typical model selection procedure [6]. LAD was found to provide the most accurate results in the shortest training time for building the reputation function. As we can see from the ROC curve in Figure 9, the LAD classifier exhibits a low false positive rate (FP%) of 0.38% and true positive rate (TP%) of 96.8%. It is was noting that these results were obtained using 10-fold cross-validation, and the detection threshold was set to 0.5. The dataset using for the evaluation contained 10,719 RRs related to 9,530 *known bad* domains. The list of *known good* domains consisted of the top 500 most popular domains according to Alexa.

We also benchmarked the reputation function on other two datasets containing a larger number of *known good* domain names. We experimented with bot the top 10,000 and top 100,000 Alexa domain names. The detection results for these experiments are as follows. When using the top 10,000 Alexa domains, we obtained a true positive rate of 93.6% and a false positive rate of 0.4% (again using 10-fold cross-validation and a detection threshold equal to 0.5). As we can see, these results are not very different from the ones we obtained using only the top 500 Alexa domains. However, when we extended our list of *known good* domains to include the top 100,000 Alexa domain names, we observed a significant decrease of the true positive rate and an increase in the false positives. Specifically, we obtained a TP% of 80.6% and a FP% of 0.6%. We believe this degradation in accuracy may be due to the fact that the top 100,000 Alexa domains include not only professionally run domains and network infrastructures, but also include less good domain names, such as file-sharing, porn-related websites, etc., most of which are not run in a professional way and have disputable reputation[1].

We also wanted to evaluate how well Notos performs, compared to static blacklists. To this end, we performed a number of experiments as follows. Given an instance of Notos trained with data collected up to July 31, 2009, we fed Notos with 250,000 distinct RRs found in DNS traffic we collected on August 1, 2009. We then computed the reputation score for each of these RRs. First, we set the detection threshold to 0.5, and with this threshold we identified 54,790 RRs that had a low reputation (lower than the threshold). These RRs where

[1] A quick analysis of the top 100,000 Alexa domains reported that about 5% of the domains appeared in the SURBL (`www.surbl.org`) blacklist, at certain point in time. A more rigorous evaluation of these results is left to future work.



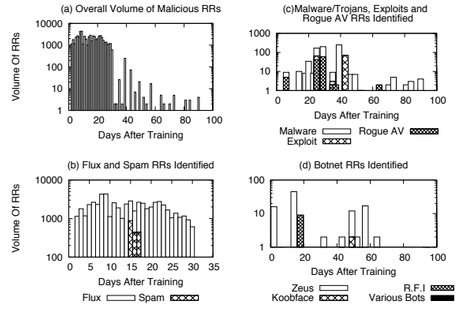



**Figure 14. Dates in which various blacklists confirmed that the RRs were malicious after Notos assigned low reputation to them on the $1^{st}$ of August.**

related to a total of 10,294 distinct domain names (notice that a domain name may map to more than one IP, and this explains the higher number of RRs). Of these 10,294 domains, 7,984 (77.6%) appeared in at least one of the public blacklists we used for comparison (see Section 4) within 60 day after August 1, and were therefore confirmed to be malicious. Figure 14(a) reports the number and date in which RRs classified as having low reputation by Notos appeared in the public blacklists. The remaining three plots (Figure 14(b), (c) and (d)), report the same results organized according to the type of malicious domains. In particular, it is worth noting that Notos is able to detect never-before-seen domain names related to the Zeus botnet several days or even weeks before they appeared in any of the public blacklists.

For the remaining 22.4% of the 10,294 domains we considered, we were not able to draw a definitive conclusion. However, we believe many of those domains are involved in some kind of more or less malicious activities. We also noticed that 7,980 or the 7,984 *confirmed bad* domain names were assigned a reputation score lower or equal to 0.15, and that none of the other non-confirmed suspicious domains received a score lower than this threshold. In practice, this means that an operator who would like to use Notos as a stand-alone dynamic blacklisting system while limiting the false positives to a negligible (or even zero) amount may fine-tune the detection threshold and set it around 0.15.

### 5.4 Discussion

This section discusses the limits of Notos, and the potential for evasion in real networks. On of the main limitations is the fact that Notos is unable to assign reputation scores for domain names with very little historic (passive DNS) information. Sufficient time and a relatively large passive DNS collection are required to create an accurate passive DNS database. Therefore, if an attacker always buys new domain names and new address space, and never reuses either resource for any other malicious purposes, Notos will not be able to accurately assign a reputation score to the new domains. In the IPv4 space, this is very unlikely to happen due to the impending exhaustion of the available address space. Once IPv6 becomes the predominant protocol, however, this may represent a problem for the statistical features we extract based on IP granularity. However, we believe the features based on BGP prefixes and AS numbers would still be able to capture the agility typical of malicious DNS hosting behavior.

As long as newly generated domain names share some network properties (e.g., IPs or BGP prefixes) with already labeled RRs, Notos will be able to assign an accurate reputation score. In particular, since network resources are finite and more expensive to renew or change, even if the domain properties change, Notos can still identify whether a domain name may be associated with malicious behavior. In addition, if a given domain name for which we want to know the reputation is not present in the passive DNS DB, we can actively probe it, thus forcing a related passive DNS entry. However, this is possible only when the domain successfully maps to a non-empty set of IPs.

Our experimental results using the top 10,000 Alexa domain names as *known good* domains, report a false positive fate of 0.4%. While low in percentage, the absolute number of false positives may become significant in those cases in which very large numbers of new domain names are fed to Notos on a daily bases (e.g., in case of deployment in a large ISP network). However, we envision our Notos reputation system to be use not as a stand-alone system, but rather in cooperation with other defense mechanisms. For example, Notos may be used in collaboration with spam-filtering system. If an email contains a link to a website whose domain name has a low reputation score according to Notos, the spam filter can increase the total spam-score of the email. However, if the rest of the email appears to be benign, the spam filter may still decide to accept the email.

During our manual analysis of (a subset of) the false positives encountered in our evaluations we were able to draw some interesting observation. We found that a number of legitimate sites (e.g., goldsgym.com) are being hosted in networks that host large volumes of malicious domain names in them. In this cases Notos will tend to penalize the reputation of this legitimate domains because they reside in a *bad neighborhood*. In time, the reputation score assigned to these domains score may change, if the administrators of the network in which the benign domain name are hosted take actions to "clean up" their networks and stop hosting bad domain names within their address space.

| Domain Name | IP | Date |
|---|---|---|
| google-bot004.cn | 213.182.197.229 | 08-15 |
| analf.net | 222.186.31.169 | 08-15 |
| pro-buh.ru | 89.108.67.83 | 08-15 |
| ammdamm.cn | 92.241.162.55 | 08-15 |
| briannazfunz.com | 95.205.116.55 | 08-15 |
| mybank-of.com | 59.125.229.73 | 08-15 |
| oc00co.com | 212.117.165.128 | 08-15 |
| avangadershem.com | 195.88.190.29 | 08-19 |
| securebizccenter.cn | 122.70.145.140 | 08-19 |
| adobe-updating-service.cn | 59.125.231.252 | 09-02 |
| 0md.ru | 219.152.120.118 | 09-19 |
| avrev.info | 98.126.15.186 | 09-27 |
| g00glee.cn | 218.93.202.100 | 09-02 |

**Table 1. Sample cases form Zeus domains detected by Notos and the corresponding days that appeared in the public BLs. All evidence information in this table were harvested from zeustracker.abuse.ch.**

| Domain Name | IP | Type | Src | Date |
|---|---|---|---|---|
| lzwn.in | 94.23.198.97 | MAL | [1] | 08-26 |
| 3b9.ru | 213.251.176.169 | MAL | [2] | 08-30 |
| antivirprotect.com | 64.40.103.249 | RAV | [3] | 09-05 |
| 1speed.info | 212.117.163.165 | CWS | [2] | 09-05 |
| spy-destroyer.com | 67.211.161.44 | CWS | [4] | 09-05 |
| free-spybot.com | 63.243.188.110 | RAV | [2] | 09-05 |
| a3l.at | 89.171.115.10 | MAL | [2] | 09-09 |
| gidromash.cn | 211.95.79.170 | BOT | [2] | 09-13 |
| iantivirus-pro.com | 188.40.52.180 | KBF | [5] | 09-19 |
| ericwanhouse.cn | 220.196.59.19 | EXP | [6] | 09-22 |
| 1165651291.com | 212.117.165.126 | RAV | [2] | 10-06 |

**Table 2. Anecdotal cases of malicious domain names detected by Notos and the corresponding days that appeared in the public BLs .**[1]: hosts-file.net, [2]: malwareurl.com, [3] siteadvisor.com, [4] virustotal.com, [5] ddanchev.blogspot.com, [6] malwaredomainlist.com

## 6 Conclusion

In this paper, we presented Notos, a dynamic reputation system for DNS. To the best of our knowledge, Notos is the first system that can assign a dynamic reputation score to any domain name in a DNS query that traverses the edge of a monitored network. Notos harvests information from multiple sources such as the DNS zone domain names belongs to, the related IP addresses, BGP prefixes, AS information and honeypot analysis to maintain up-to-date DNS information about legitimate and malicious domain names. Based on this information, Notos uses automated classification and clustering algorithms to model network and zone behaviors of legitimate and malicious domains, and then applies these models to compute a reputation score for a (new) domain name.

Our evaluation using real-world data, which includes traffic from large ISP networks, demonstrates that Notos is highly accurate in identifying new malicious domains in the monitored DNS query traffic, with a true positive rate of 96.8% and false positive rate of 0.38%. In addition, Notos is capable of identifying these malicious domain weeks or even months before they appear in public blacklists, thus enabling proactive security countermeasures against cyber attacks.

## 7 Acknowledgments

We thank Steven Gribble, our shepherd, for helping us to improve the quality of the final version of this paper, and the anonymous reviewers for their constructive comments. We also thank Gunter Ollmann and Robert Edmonds for their valuable comments. Additionally, we thank the Internet Security Consortium Security Information Exchange project (ISC@SIE) for providing portion of the DNS data used in our experiments.

This material is based upon work supported in part by the National Science Foundation under grant no. 0831300, the Department of Homeland Security under contract no. FA8750-08-2-0141, the Office of Naval Research under grants no. N000140710907 and no. N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Department of Homeland Security, or the Office of Naval Research.

## References

[1] D. Anderson, C. Fleizach, S. Savage, and G. Voelker. Spamscatter: Characterizing internet scam hosting infrastructure. In *Proceedings of the USENIX Security Symposium*, 2007.

[2] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[3] Internet Systems Consortium. SIE@ISC : Security Information Exchange. `https://sie.isc.org/`, 2004.

[4] A. Dinaburg, R. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM CCS*, 2008.

[5] SORBS DNSBL. Fighting spam by finding and listing Exploitable Servers. `http://www.us.sorbs.net/`, 2007.

[6] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2000.

[7] M. Felegyhazi, C. Keibich, and V. Paxson. On the potential of proactive domain blacklisting. In *Third USENIX LEET Workshop*, 2010.

[8] S. Garera, N. Provos, M. Chew, and A. Rubin. A framework for detection and measurement of phishing attacks. In *Proceedings of the ACM WORM*. ACM, 2007.

[9] B. Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *ACM CCS 09*, New York, NY, USA, 2009. ACM.

[10] T. Holz, C. Gorecki, K. Rieck, and F. Freiling. Measuring and detecting fast-flux service networks. In *Proceedings of NDSS*, 2008.

[11] T. Hothorn and B. Lausen. Double-bagging: Combining classifiers by bootstrap aggregation. *Pattern Recognition*, 36(6):1303–1309, 2003.

[12] P. Mockapetris. Domain names - concepts and facilities. `http://www.ietf.org/rfc/rfc1034.txt`, 1987.

[13] P. Mockapetris. Domain names - implementation and specification. `http://www.ietf.org/rfc/rfc1035.txt`, 1987.

[14] OPENDNS. OpenDNS — Internet Navigation And Security. `http://www.opendns.com/`, 2010.

[15] P. Porras, H. Saidi, and V. Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. `http://mtc.sri.com/Conficker/`, 2009.

[16] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *USENIX NSDI*, 2010.

[17] D. Plonka and P. Barford. Context-aware clustering of DNS query traffic. In *Proceedings of the 8th IMC*, Vouliagmeni, Greece, 2008. ACM.

[18] The Spamhaus Project. ZEN - Spamhaus DNSBLs. `http://www.spamhaus.org/zen/`, 2004.

[19] R. Perdisci, I. Corona, D. Dagon, and W. Lee. Detecting malicious flux service networks through passive analysis of recursive DNS traces. In *Proceedings of ACSAC*, Honolulu, Hawaii, USA, 2009.

[20] P. Royal. Analysis of the kraken botnet. `http://www.damballa.com/downloads/r_pubs/KrakenWhitepaper.pdf`, 2008.

[21] S. Hao, N. Syed, N. Feamster, A. Gray and S. Krasser. Detecting spammers with SNARE: Spatio-temporal network-level automatic reputation engine. In *Proceedings of the USENIX Security Symposium*, 2009.

[22] S. Shevchenko. Srizbi Domain Generator Calculator. `http://blog.threatexpert.com/2008/11/srizbis-domain-calculator.html`, 2008.

[23] K. Sato, K. Ishibashi, T. Toyono, and N. Miyake. Extending black domain name list by using co-occurrence relation between dns queries. In *Third USENIX LEET Workshop*, 2010.

[24] S. Sinha, M. Bailey, and F. Jahanian. Shades of grey: On the effectiveness of reputation-based blacklists. In *3rd International Conference on MALWARE*, 2008.

[25] The Honeynet Project & Research Alliance. Know Your Enemy: Fast-Flux Service Networks. `http://old.honeynet.org/papers/ff/fast-flux.html`, 2007.

[26] URIBL. Real time URI blacklist. `http://uribl.com`.

[27] F. Weimer. Passive DNS replication. In *Proceedings of FIRST Conference on Computer Security Incident*, Handling, Singapore, 2005.

[28] Z. Qian, Z. Mao, Y. Xie and F. Yu. On network-level clusters for spam detection. In *Proceedings of the USENIX NDSS Symposium*, 2010.

[29] B. Zdrnja, N. Brownlee, and D. Wessels. Passive monitoring of DNS anomalies. In *Proceedings of DIMVA Conference*, 2007.

[30] Zeus Tracker. Zeus IP & domain name block list. `https://zeustracker.abuse.ch`, 2009.

[31] J. Zhang, P. Porra, and J. Ullrich. Highly predictive blacklisting. In *Proceedings of the USENIX Security Symposium*, 2008.

# Scantegrity II Municipal Election at Takoma Park: The First E2E Binding Governmental Election with Ballot Privacy

Richard Carback, *UMBC CDL*
David Chaum
Jeremy Clark, *University of Waterloo*
John Conway, *UMBC CDL*
Aleksander Essex, *University of Waterloo*
Paul S. Herrnson, *UMCP CAPC*
Travis Mayberry, *UMBC CDL*
Stefan Popoveniuc
Ronald L. Rivest, *MIT CSAIL*
Emily Shen, *MIT CSAIL*
Alan T. Sherman, *UMBC CDL*
Poorvi L. Vora, *GW*

## Abstract

On November 3, 2009, voters in Takoma Park, Maryland, cast ballots for the mayor and city council members using the Scantegrity II voting system—the first time any *end-to-end (E2E)* voting system with ballot privacy has been used in a binding governmental election. This case study describes the various efforts that went into the election—including the improved design and implementation of the voting system, streamlined procedures, agreements with the city, and assessments of the experiences of voters and poll workers.

The election, with 1728 voters from six wards, involved paper ballots with invisible-ink confirmation codes, instant-runoff voting with write-ins, early and absentee (mail-in) voting, dual-language ballots, provisional ballots, privacy sleeves, any-which-way scanning with parallel conventional desktop scanners, end-to-end verifiability based on optional web-based voter verification of votes cast, a full hand recount, thresholded authorities, three independent outside auditors, fully-disclosed software, and exit surveys for voters and pollworkers.

Despite some glitches, the use of Scantegrity II was a success, demonstrating that E2E cryptographic voting systems can be effectively used and accepted by the general public.

## 1 Introduction

The November 2009 municipal election of the city of Takoma Park, Maryland marked the first time that anyone could verify that the votes were counted correctly in a secret ballot election for public office without having to be present for the entire proceedings. This article is a case study of the Takoma Park election, describing what was done—from the time the Scantegrity Voting System Team (SVST) was approached by the Takoma Park Board of Elections in February 2008, to the last cryptographic election audit in December 2009—and what was learned. While the paper provides a simple summary of survey results, the focus of this paper is not usability but the engineering process of bringing a new cryptographic approach to solve a complex practical problem involving technology, procedures, and laws.

With the Scantegrity II voting system, voters mark optical scan paper ballots with pens, filling the oval for the candidates of their choice. These ballots are handled as traditional ballots, permitting all the usual automated and manual counting, accounting, and recounting. Additionally, the voting system provides a layer of integrity protection through its use of invisible-ink confirmation codes. When voters mark ballot ovals using a decoder pen, confirmation codes printed in invisible ink are revealed. Interested voters can note down these codes to check them later on the election website. The codes are generated randomly for each race and each ballot, and hence do not reveal the corresponding vote. A final tally can be computed from the codes and the system provides a public digital audit trail of the computation.

Election audits in Scantegrity II are not restricted to privileged individuals and can be performed by voters and other interested parties. Developers and election authorities are unable to significantly falsify an election outcome without an overwhelming probability of an audit failure [8]. The other side of the issue of integrity, also solved by the system, is that false claims of impropriety in the recording and tally of the votes are readily revealed to be false. [1]

All the software used in the election—for ballot authoring, printing, scanning and tally—was published well in advance of the election as commented, buildable source code, which may be a first in its own right. Moreover, commercial off-the-shelf scanners were adapted to receive ballots in privacy sleeves from voters, making the

[1] Note that a threat present and not commonly addressed in paper ballot systems is that additional marks could be added to ballots by those with special access. Such attacks are made more difficult by Scantegrity II.

overall system relatively inexpensive.

Despite several limitations of the implementation, we found that the amount of extra work needed by officials to use Scantegrity II while administering an election is acceptable given the promise of improved voter satisfaction and indisputability of the outcome. Indeed, discussions are ongoing with the Board of Elections of the city regarding continued use of the system in future elections.

Another observation from the election is that the election officials and voters surveyed seemed to appreciate the system. Since voters who do not wish to verify can simply proceed as usual, ignoring the codes revealed in the filled ovals, the system is least intrusive for these voters. Those voters who did check their codes, and even many who did not, seem to appreciate the opportunity.

This paper describes the entire process of adapting the Scantegrity II system to handle the Takoma Park election, including the agreement with the city, printing the special ballots with invisible-ink confirmation codes, actually running the election, and verifying that the election outcome was correct.

**Organization of this case study** The next section provides an overview of related work in this area, summarizing previous experiments with Scantegrity II and other E2E systems in practical settings.

Section 3 describes in more detail the setting for the election: giving details about Takoma Park and their election requirements. Section 4 gives more details of the Scantegrity II voting system, including a description of how one can "audit" an election. Section 5 provides an overview of the implementation of the voting system for the November 3, 2009 Takoma Park municipal election, including the scanner software, the cryptographic back-end, and the random-number generation routines.

Section 6 gives a chronological presentation and timeline of the steps taken to run the November election, including the outcome of the voter verification and the audits. It also gives the results of the election, with some performance and integrity metrics. Section 7 reports some results of the exit surveys taken of voters and pollworkers.

Section 8 discusses the high-level lessons learned from this election. Section 9 provides some conclusions, acknowledgements, and disclosures required by the program committee.

## 2 Related Work

Chaum was the first to propose the use of cryptography for the purpose of secure elections [5]. This was followed by almost two decades of work in improving security and privacy guarantees (for a nice survey, see Adida [1]), most recently under the rubric of *end-to-end* voting systems. These voting system proposals provide integrity (any attempt to change the tally can be caught with very high probability by audits which are not restricted to privileged individuals) and ballot secrecy.

The first of these proposals include protocols by Chaum [6] and Neff [19], which were implemented soon after (Chaum's as *Citizen-Verified Voting* [16] and Neff's by VoteHere). Several more proposals with prototypes followed: *Prêt à Voter* [10], *Punchscan* [21, 15], the proposal of Kutylowski and Zagórski [18] as *Voting Ducks*, and Simple Verifiable Voting [4] as *Helios* [2] and *VoteBox* [24].

Making end-to-end systems usable in real elections has proven to be challenging. We are aware of the following previous binding elections held using similar verification technology: the *Punchscan* elections for the graduate students' union of the University of Ottawa (2007) and the Computer Professionals for Social Responsibility (2007); the Rijnland Internet Election System (RIES) public elections in the Netherlands in 2004 and 2006; the *Helios* elections of the Recteur of Université Catholique de Louvain [3] (2009) and the Princeton undergraduate student government election (2009), as well as a student election using *Prêt à Voter*.

Only the RIES system has been used in a governmental election; however, it is meant for remote (absentee) voting and, consequently, does not offer strong ballot secrecy guarantees. For this reason, it has been recommended that the RIES system not be used for regular public elections [17, 20]. *Helios* is also a remote voting system, and offers stronger ballot secrecy guarantees over RIES. The *Punchscan* elections were the closest to this study, but they did not rise to the level of public elections. They did not have multiple ballot styles, the users of the system were not a broad cross-segment of the population as in Takoma Park, the system implementors were deeply involved in administering the elections, and no active auditors were established to audit the elections. To date, this study is the most comparable use case of E2E technology to that of a typical optical scan election.

The case study reported here is based on a series of systems successively developed, tested, and deployed by a team of researchers included among the present authors originating with the *Punchscan* system. Although it used paper ballots, the *Punchscan* system did not allow manual recounts, a feature that the team recognized as needing to be designed into the next generation of systems. The result was *Scantegrity* [9], which retained hand-countable ballots, and was tested in a number of small elections. With *Scantegrity*, however, it was too easy to trigger an audit that would require scrutiny of the physical ballots. The *Scantegrity II* system [7, 8], deployed in Takoma Park, was a further refinement to address this problem by allowing a public statistical test of whether voter complaints actually reflect a discrepancy or whether they are without basis. Note: in the rest of the paper, "Scantegrity" refers to the voting team or to the Scantegrity II voting system; which one is typically easily determined from context.

As part of the Scantegrity agreement with Takoma Park (see section 3), a "mock election" [26] was held in April 2009 to test and demonstrate feasibility of the Scantegrity system during Takoma Park's annual Arbor day celebration. Volunteer voters voted for their favorite tree. A number of revisions and tweaks to the Scantegrity system were made as a result of the mock election, including: ballot revisions (no detachable chit, but instead a separate voter verification card), pen revisions (two-ended, with different sized tips), scanner station revisions (better voter flow, no monitor, two scanners), privacy sleeve (no lock, no clipboard, folding design, feeds directly into scanner), and confirmation codes (three decimal digits).

## 3 The Setting

For several reasons, the implementation of voting systems is a difficult task. Most voting system users—*i.e.* the voters—are untrained and elections happen infrequently. Voter privacy requirements preclude the usual sorts of feedback and auditing methods common in other applications, such as banking. Also, government regulations and pre-existing norms in the conduct of elections are difficult to change. These issues can pose significant challenges when deploying new voting systems, and it is therefore useful to understand the setting in which the election took place.

**About Takoma Park** The city of Takoma Park is located in Montogomery County, Maryland, shares a city line with Washington, D.C, and is governed by a mayor and a six-member City Council. The city has about 17,000 residents[2] and almost 11,000 registered voters [27, pg. 10]. A seven-member Board of Elections conducts local elections in collaboration with the City Clerk. In the past, the city has used hand counts and optical scan voting, as well as DREs for state elections.

The Montgomery County US Census Update Data of 2005 provides some demographic information about the city. Median household income in 2004 was $48,675. The percentage of households with computers was 87.4%, and about 32% of Takoma Park residents above the age of twenty-five had a graduate, professional or doctoral degree. It is an ethnically diverse city: 45.8% of its residents identify their race as "White," 36.3% as "Black," 9.7% as "Asian or Pacific Islander" and 8.2% as "Other" (individuals of Hispanic origin form the major component of this category). Further, 44.4% of its households have a foreign-born head of household or spouse, and 44.8% of residents above the age of five spoke a language other than English at home.

**Instant Runoff Voting (IRV)** Takoma Park has used IRV in municipal city elections since 2006. IRV is a ranked choice system where each voter assigns each candidate a rank according to her preferences. The rules[3] used by Takoma Park (and the Scantegrity software) for counting IRV ballots are relatively standard, so we omit further discussion for lack of space.

**Agreement with the City** As with any municipal government in the US, Takoma Park is allowed to choose its own voting system for city elections. For county, state, and federal elections, it is constrained by county, state, and federal election laws.

Takoma Park and the SVST signed a Memorandum of Understanding (MOU), in which the SVST agreed to provide equipment, software, training assistance, and technical support. The City of Takoma Park agreed to provide election-related information on the municipality, election workers, consumable materials, and perform or provide all other election duties or materials not provided by us. No goods or funds were exchanged.

According to the MOU, if approved by the city council, the election was to be conducted in compliance with all applicable laws and policies of the city. This included using Instant Runoff Voting as defined by the City of Takoma Park Municipal Charter.

The SVST also agreed to pursue an accessible ballot-marking device for the election, but was later relieved of satisfying this requirement. Unfortunately, Scantegrity is not yet fitted with a voter interface for those with visual or motor disabilities, and accessible user interfaces were also not used in Takoma Park's previous optical scan elections.

**Timeline** Scantegrity was approached by the Takoma Park Board of Elections in late February 2008, and, after considering other voting systems, the Board voted to recommend a contract with Scantegrity in June 2008. Following a public presentation to the City Council in July 2008, the MOU was signed in late November 2008, about nine months after the initial contact.

[2] See `http://www.takomaparkmd.gov/about.html`.

[3] For the exact laws used by Takoma Park, see page 22 of `http://www.takomaparkmd.gov/code/pdf/charter.pdf`. Section (f), concerning eliminating multiple candidates, was used in our implementation for tie-breaking only.

The SVST held an open workshop in February 2009 to discuss the use of Scantegrity in both the mock and real elections. This workshop was held at the Takoma Park Community Center and was attended by Board of Election members, the City Clerk, current members (and a retired member) from the Montgomery County Board of Elections, as well as a representative each from the Pew Trust and FairVote. Following the mock election in April 2009, the SVST proposed a redesigned system taking into consideration feedback from voters and poll workers (through surveys) and the Board of Elections. The Board voted to recommend use of the redesigned system in July 2009; this was made official in the city election ordinance in September 2009. [4] Beginning around June 2009, a meeting with representatives of the SVST was on the agenda of most monthly Board of Election meetings. Additionally, SVST members met many times with the City Clerk and the Chair of the Board of Elections to plan for the election.

The final list of candidates was available approximately a month before the election, on October 2. The Scantegrity *meetings* initializing the data and ballots were held in October (see Section 6), as was a final workshop to test the system. Absentee ballots were sent out by the City Clerk in the middle of October. The SVST delivered ballots to the City Clerk in late October, and early voting began almost a week before the election, on October 28. Poll worker training sessions were held by the city on October 28 and 31, and polling on November 3, 2009, from 7 am to 8 pm. The final Scantegrity audits were completed on 17 December 2010; all auditors were of the opinion that the election outcomes were correct (for details see section 6).

## 4 Scantegrity Overview

In this section, we give an overview of the Scantegrity system. For more detailed descriptions, see [7, 8].

**Voter Experience** At a high level, the voter experience is as follows. First, a voter checks in at the polling place and receives a Scantegrity ballot (See Figure 2) with a privacy sleeve. The privacy sleeve is used to cover the ballot and keep private the contents of the ballot. Inside the voting booth, there is a special "decoder pen" and a stack of blank "voter verification cards." The voter uses the decoder pen to mark the ballot. As on a conventional optical scan ballot, she fills in the bubble next to each of her selections. Marking a bubble with the decoder pen simultaneously leaves a dark mark inside the bubble and reveals a previously hidden confirmation code printed in invisible ink.

If the voter wishes to verify her vote later on the election website, she can copy her ballot ID and her revealed confirmation codes onto a voter verification card. She keeps the verification card for future reference. She then takes her ballot to the scanning station and feeds the ballot into an optical scanner, which reads the ballot ID and the marked bubbles.

If a voter makes a mistake, she can ask a poll worker to replace her ballot with a new one. The first ballot is marked "spoiled," and its ballot ID is added to the list of spoiled ballot IDs maintained by the election judges.

The voter can verify her vote on the election website by checking that her revealed confirmation codes and ballot ID have been posted correctly. If she finds any discrepancy, the voter can file a complaint through the website, within a complaint period. When filing a complaint, the voter must provide the confirmation codes that were revealed on her ballot as evidence of the validity of the complaint.

**Ballots** The Scantegrity ballot looks similar to a conventional optical scan ballot (see Figure 2 for a sample ballot used in the election). It contains a list of the choices and bubbles beside each choice. Marking a bubble reveals a random 3-digit confirmation code.

**Confirmation Codes** The confirmation codes are unique within each contest on each ballot, and are generated independently and uniformly pseudorandomly. The confirmation code corresponding to any given choice on any given ballot is hidden and unknown to any voter until the voter marks the bubble for that choice.

**Digital Audit Trail** Prior to the election, a group of election trustees secret-share a seed to a pseudorandom number generator (PRNG). The trustees then input their shares to a trusted workstation to generate the pseudorandom confirmation codes for all ballots, as well as a set of tables of cryptographic commitments to form the digital audit trail. These tables allow individual voters to verify that their votes have been included in the tally, and allow any interested party to verify that the tally has been computed correctly, without revealing how any individual voter voted.

**Auditing** After the election, any interested party can audit the election by using software to check the correctness of the data and final tally on the election website. Additionally, at the polling place on the day of the election, any interested party can choose to audit the printing of the ballots. A print audit consists of marking all of the bubbles on a ballot, and then either making a photocopy of the fully-marked ballot or copying down all of the revealed confirmation codes. The ballot ID is recorded by an election judge as audited. After the election, one can check that all of the confirmation codes on the audited ballot, and their correspondence with ballot choices, are posted correctly on the election website.

[4] See `http://www.takomaparkmd.gov/clerk/agenda/items/2009/090809-3.pdf`, section 2-D, page 2.



## 5 Implementation

The election required a cryptographic *backend*, a *scanner*, and a *website*. These 3 components form the basic election system and their interaction is described in Figure 1. In addition, Takoma Park required software to resolve write-in candidate selections and produce a formatted tally on election night.

Scantegrity protects against manipulation of election results and maintains, but does not improve, the privacy properties of optical scan voting systems that use serial numbers. To compromise voter privacy using Scantegrity features, an attacker must associate receipts to voters and determine what confirmation numbers are associated to each candidate. This is similar to violating privacy by other means; for example, an attacker could compromise the scanner and determine the order in which voters used the device, or examine physical records and associate serial numbers to voters. The scanner and backend components protect voter privacy, but the website and the write-in candidate resolver do not because they work with public information only.

Each component is written in Java. We describe the implementation and functions of each one in the following sections.

**Backend** The cryptographic backend that provides the digital audit trail is a modified version of the Punchscan backend [21]. This backend is written in Java 1.5 using the BouncyCastle cryptography library. [5] Key management in the Punchscan backend is handled by a simple threshold [25] cryptosystem that asks for a username and password from the election officials.

We chose the Punchscan backend over newer proposals [7] because it had already been implemented and tested in previous elections [13, 28]. At the interface between the Scantegrity frontend and the Punchscan backend, as described in [23], the permutations used by Punchscan are matched to a permutation of precomputed confirmation codes for Scantegrity that correspond to the permutation of codes printed on the ballot.

The Punchscan backend uses a two-stage mix process based on cryptographic commitments published before the election. Each mix, the *left mix* and the *right mix*, takes marked positions as input, shuffles the ballots, and reorders each marked position on each ballot according to a prescribed (pre-committed) permutation. The result is the set of cleartext votes, where position 0 corresponds to candidate 0, 1 to 1, etc. Between the two mixes, for example, position 0 may in fact correspond to candidate 5, depending on the permutation in the *right mix*.

The Punchscan backend partitions [22] each contest such that each contest is treated as an independent election with a separate set of commitments. In the case of Takoma Park, each ward race and the mayor's race are treated as separate elections. (The announcement of separate mayoral race vote counts for each ward is required by Takoma Park). The scanner is responsible for creating the input files for each individual election.

Election officials hold a series of meetings using the backend to conduct an election. Before the election, during *Meeting 1* (Initialization), they choose passwords that are shares of a master key that generates all other data for the election in a deterministic fashion. After each meeting, secret data (such as the mapping from confirmation codes to candidates) is erased from the hard drive and regenerated from the passwords when it is needed again. In Meeting 1 the backend software creates a digital audit trail by committing to the Punchscan representation of candidate choices and to the *mixset*: the left and right mix operations for each ballot. Later, during *Meeting 2* (Pre-Election Audit), the backend software responds to an audit of the trail demonstrating that the mixset decrypts ballots correctly. At this time, the backend also commits to the Scantegrity front-end, consisting of the linkage between the Scantegrity front-end and its Punchscan backend used for decryption.

After the election, election officials run *Meeting 3* (Results), publishing the election results and the voted confirmation numbers. For the purposes of the tally audit, the system also publishes the outputs of the left and right mixes. In *Meeting 4* (Post-Election Audit), officials respond to the challenges of the tally computation audit. Either the entire *left mix* or the entire *right mix* operations are revealed, and the auditor checks them against data published in *Meeting 3*.

The *Meeting 4* audit catches, with probability one half, a voting system that cheats in the tally computation. To provide higher confidence in the results, the backend creates multiple sets of left and right mixes; in Takoma Park, we created 40 sets for each election, 20 of which were audited. Given 2 contests per ballot and 40 sets of left and right mixes, there are a total of 160 commitments per ballot in the audit trail, in addition to a commitment per contestant per ballot for each confirmation number (15-18, depending on the Ward).

The implementation uses two classes of "random" number sources. The first is used to generate the dig-

[5] `http://www.bouncycastle.org`

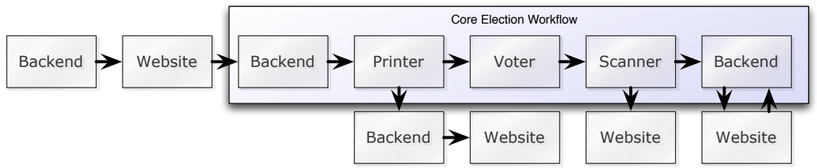



Figure 1: **Election Workflow**. The core election work flow in Scantegrity is similar to an optical scan election: a software backend creates ballot images that are printed, used by voters, and scanned. The results are fed to the backend which creates the tally. The audit capacity is provided by 3 extra steps: (1) create the initial digital audit trail and audit a portion of it, (2) audit the ballots to ensure correctness when printing, and (3) audit the final tally.

ital audit trail, and the second is used for auditing the trail. Both types of sources must be unpredictable to an adversary, and we describe each in turn.

***Digital Audit Trail*** The Punchscan backend generates the mixes and commitments using entropy provided by each election official during initialization of the threshhold encryption. This provided a "seed" for a pseudorandom number generator (based on the SHA256 hash function).

We also used this random source to generate the confirmation numbers when changing the Punchscan backend to support Scantegrity. Unfortunately, we introduced an error in the generation when switching from alphanumeric to numeric confirmation numbers as a result of findings in the Mock election (see Section 2). This resulted in approximately 8.5 bits of entropy as opposed to the expected 10 bits. We discovered this error after we started printing and it was too late to regenerate the audit trail.

The error increased the chance that an adversary could guess an unseen confirmation code to approximately one in 360 rather than the intended one in 1000; a small decrease in the protection afforded against malicious voters trying to guess unseen codes in order to discredit the system.

***Auditing*** Random numbers are needed to generate challenges for the various auditing steps (print audit, randomized partial checking). These numbers should be unpredictable in advance to an adversary. They should also be "verifiable" after the fact as having come from a "truly random" source that is not manipulable by an adversary.

We chose to use the closing prices of the stocks in the Dow Jones Industrial Average as our verifiable but unpredictable source to seed the pseudorandom number generator (the use of stock prices for this purpose was first described in [11]). These prices are sufficiently unpredictable for our purposes, yet verifiable after the fact. However, it turns out that post-closing "adjustments" can sometimes be made to the closing prices, which can make these prices less than ideal for our purposes in terms of verifiability.

**Scanner Software** The original intent of Scantegrity was to build on top of an existing optical scan system. There was no pre-existing optical scan system in use at Takoma Park, so we implemented a simple system using EeePC 900 netbooks and Fujitsu 6140 scanners.

The scanning software is written in Java 1.6. It uses a bash shell script to call the SANE scanimage program [6] and polls a directory on the filesystem to acquire ballot images. Once an image is acquired it uses circular alignment marks to adjust the image, reads the barcode using the ZXing QRCode Library, [7] and uses a simple threshold algorithm to determine if a mark is made on the ballot.

Individual races on each ballot are identified by ward information in the barcode, which is non-sequential and randomly generated. The ballot id in the barcode and the web verification numbers on each ballot are different numbers, and the association between each number type is protected by the backend system. Write-in candidate areas, if that candidate is selected by the voter, are stored as clipped raw images with the ballot scan results. Ballot scan results are stored in a random location in a memory mapped file.

The current implementation of the scanning software does not protect data in transit to the backend, which poses a risk for denial of service. Checking of the correctness of the scanner is done through the Scantegrity audit. The data produced by the scanner does not compromise voter privacy, but—assuming an attacker could intercept scanner data—voter privacy could be compromised at the scanner through unique write-in candidates on the ballot, through a compromised scanner, by bugs in the implementation, or by relying on the voter to make readable copies of the barcode to get a ballot id.

[6] http://www.sane-project.org/
[7] http://code.google.com/p/zxing/



**Tabulator/Write-In Software** At the request of Takoma Park we created an additional piece of software, the Election Resolution Manager (ERM), that allows election judges to manually determine for each write-in vote what candidate the vote should be counted toward. The other responsibility of the ERM is to act as part of the backend. It collates data from each scanner and prepares the input files for the backend.

To resolve write-ins with this software, the user cycles through each image, and either types in the name of the intended candidate or selects the name from a list of previously identified candidates composed of the original candidates and any previously typed candidate names. The user is not shown the whole ballot, so he does not know what the other selections are on that ballot, or what rank the write-in was given. We call this process *resolving* a vote because the original vote is changed from the generic "Write-In" candidate to the candidate that was intended by the voter. The ERM produces a PDF of each image, the candidate selection for that image, and a unique number to identify the selection.

Scantegrity handles write-in candidates just like other optical scan systems by treating the write-in position as a candidate. Therefore, the backend does not know how each write-in position was resolved, and two results records are created: one with write-in resolution provided by the ERM, and one without write-in resolution provided by the backend.

To check the additional record generated by the ERM, an observer reduces the resolved results record and verifies that the set of resolved ballots is the same as the set of unresolved ballots. To audit that the judges chose the correct candidates for each write-in, the observer refers to the PDF generated during write-in resolution. The PDF allows the observer to reference each resolved ballot entry in the resolved results file and verify that the image was properly transcribed.

One caveat of this approach is that if a write-in candidate wins, a malicious authority could modify these images to change results, but could not deny that the write-in position had received a winning number of votes. This situation would require additional procedures to verify the write-ins (e.g. a hand count, and/or careful audit of the transcriptions by each judge).

**Website** Beyond communicating the election outcome itself, the role of the election website is to serve as a "bulletin board" (BB) to broadcast the cryptographic audit data set (i.e., cryptographic commitments, responses to audit challenges, etc). In addition, voters can use this website to check their receipts, and file a dispute if the receipt is misreported. We provided an implementation with these features written in Java 1.6. It used the Stripes Framework [8] and an Apache Derby database backend. [9] In practice, we only used part of this implementation.

Originally, our plan was to have Takoma Park host the website, but officials chose a hybrid approach where they hosted election information and results. That website would link to our server to provide a receipt checking tool and audit data. After the election, officials would provide us with a copy of the public data files to publish. This decision caused a number of changes to our approach.

We decided to only use the receipt checking code from the implementation, and, to make downloading more convenient for auditors, post all election data on our publicly available subversion repository. [10] Additionally, both auditors agreed to mirror the data.

A primary security requirement for the Scantegrity BB is to provide authenticated broadcast communication from election officials to the public. We met this requirement with digital signatures. A team member (Carback) created signed copies of each file with gnupg [11] using his public key from May 28, 2009.

Without authenticated communication, it would be impossible to prove if different results were provided to different people. Our specific approach to the website requires observers to verify signatures and check with each other if they receive identical copies of the data (and verify the consistency of the signatures over time). Our auditors, Adida and Zagorski, performed these actions, but we do not know the extent of this communication otherwise. As usual with our approach to Scantegrity, we are enabling **detection** of errors (genuine or malicious).

There are several potential threats to the bulletin board model–we will briefly enumerate some of them. At a high level, threats pertain primarily to misreporting of results, or to voter identification. With regard to results reporting, an adversary may attempt to misreport results by substituting actual election data with false data. In the event that all parties verify signatures of information they receive, and check consistency with the signed files, incorrect confirmation codes on the bulletin board would be detected by voters, and incorrect computation of the tally by anyone checking the tally computation audit. If the voter checking confirmation codes does not check consistency with the rest of the bulletin board (by, for example, downloading the bulletin board data, checking all the signatures and checking that his or her confirmation code is also correctly noted in the entire bulletin board data) he or she may be deceived into believing their ballot was accurately recorded and counted. Similarly, if

[8] http://www.stripesframework.org/
[9] http://db.apache.org/derby/
[10] http://scantegrity.org/svn/data/takoma-nov3-2009/
[11] http://www.gnupg.org/

the various signatures are not cross checked across individuals or observed over time, an adversary may replace the confirmation codes after they have been checked, or send different ones to voters and to auditors. An adversary may also attempt an identification attack, whereby the objective is to link voter identities with receipt data, such as by recording IP addresses of voters who check their receipts.

## 6 The Election

In this section, we describe the election as events unfold chronologically over time.

### 6.1 Preparations

Preparations for the election include running the first 2 backend meetings, and creating the ballot.

**Independent Auditors** The Board of Elections requested cryptographers Dr. Ben Adida (Center for Research on Computation and Society, Harvard University) and Dr. Filip Zagórski (Institute of Mathematics and Computer Science, Wroclaw University of Technology, Poland) to perform independent audits of the digital data published by Scantegrity in general, and of the tally computation in particular. Dr. Adida [12] and Dr. Zagórski [13] maintained websites describing the audits and the results of the audits, and Dr. Adida also blogged the audit. [14] Before the election, Dr. Adida pointed out several instances when the Scantegrity information was insufficient; Scantegrity documentation was updated as a result.

The Board of Elections also requested Ms. Lillie Coney (Associate Director, Electronic Privacy Information Center and Public Policy Coordinator for the National Committee for Voting Integrity (NCVI)) to perform print audits on Election Day. Ms. Coney chose ballots at random through the day, exposed the confirmation codes for all options on the ballot, and kept these with her until after the end of the complaint period, when Scantegrity opened commitments to all unvoted and unspoiled ballots (and hence to all ballots she had audited). Ms. Coney then checked that the correspondence between codes and confirmation numbers on her ballots matched those on the website.

Both tasks, of print audits and digital data audits, can be performed by voters. Digital data audits can also be performed by any observers. In future elections, when the general population and Takoma Park voters are more familiar with end-to-end elections, it is anticipated that voters (and, in particular, candidate representatives) will perform such audits.

[12] http://sites.google.com/site/takomapark2009audit/

[13] http://zagorski.im.pwr.wroc.pl/scantegrity/

[14] http://benlog.com/articles/category/takoma-park-2009/

**Meeting 1** Four election officials (the City Clerk, the Chair, Vice Chair and a member of the Board of Elections: Jessie Carpenter, Anne Sergeant, Barrie Hofmann and Jane Johnson, respectively) were established as election trustees in *Meeting 1*, held on October 12 2009.

It was explained to the trustees that, through their passwords, they would generate the confirmation codes and share the secret used to tally election results. Further, it was explained that, without more than a threshold of passwords, the election could not be tallied by Scantegrity, and that if a threshold number of passwords was not accessible (if they were forgotten, for example, or trustees were unavailable due to sickness) the only available counts would be manual counts. A threshold of two trustees was determined based on anticipated availability of the officials, and it was explained that two trustees could collude to determine the correspondence between confirmation numbers and codes, and hence that each trustee should keep her password secret.

The trustees generated commitments to the decryption paths for each of 5000 ballots per ward (for six wards). Scantegrity published the commitments on October 13 2009 at 12:13am.

**Meeting 2** In *Meeting 2*, held on October 14, 2009, trustees used Scantegrity-written code to respond to challenges generated using stock market data at closing on October 14. Half of the ballot decryption paths committed to in *Meeting 1* were opened. Additionally, trustees constructed ballots (associations between candidates and confirmation codes) at this meeting, and generated commitments to them. Scantegrity published the stock market data, the challenges, and the responses.

**Ballot Design** The ballot used for the 2009 election was based on ballots used for the 2007 election. We made the conscious choice to modify (as little as possible) a design already used successfully in a past election, and not to use the ballot we had designed for the mock election. The main reason for reusing the ballot design was that it would be familiar to voters. The ballot was required to contain instructions in both English and Spanish: marking instructions, instructions for write-ins, instructions for IRV and any Scantegrity-related instructions (see Figure 2).

**Printing Ballots** We use "invisible" ink to print the marking positions that reveal confirmation codes to voters. We used refillable inkjet cartridges in multiple color

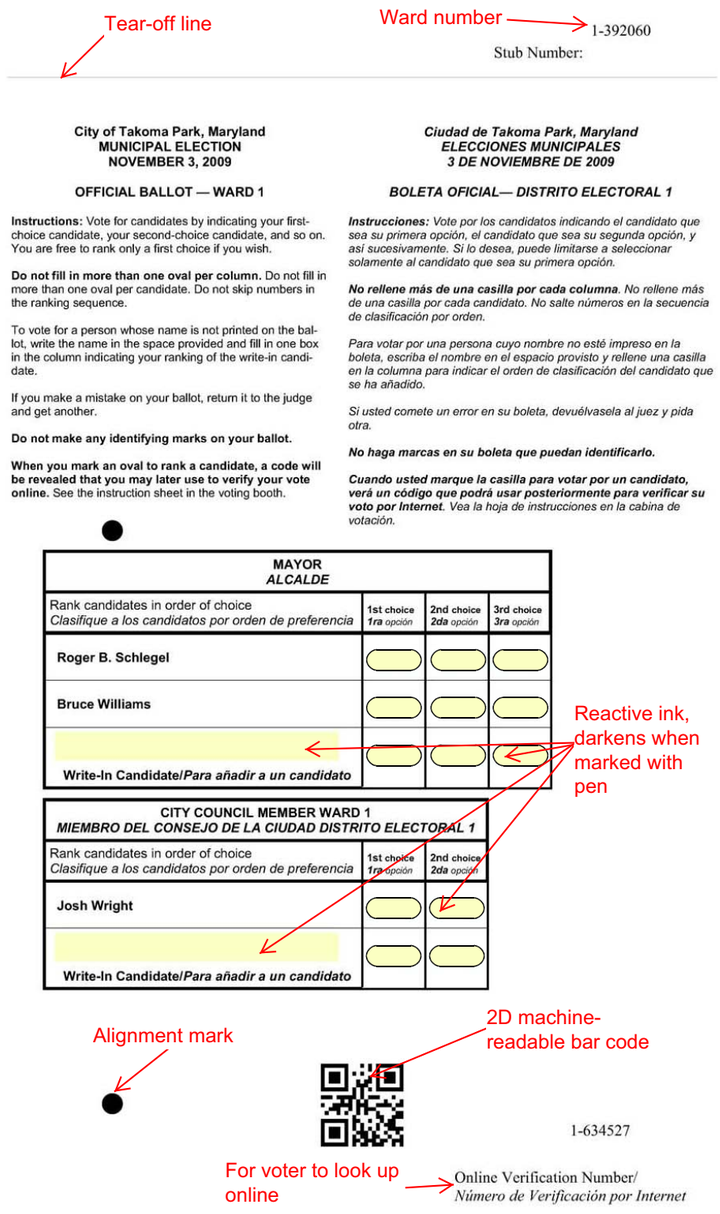



Figure 2: An unmarked Takoma Park 2009 ballot for Ward 1 showing instructions in Spanish and English, the options, the circular alignment marks, the 2D barcode, the ballot serial number (on the stub, meant for poll workers to keep track of the number of ballot used) and the online verification number (for voters to check their codes). The true ballot was printed on legal size paper and was hence larger than shown.

positions of an Epson R280 printer to print confirmation codes. The ink is not actually invisible, but looks like a yellow bubble before marking and a dark bubble with light yellow codes after marking. [15]

We initially began printing with 6 printers, but they proved unreliable. It was our expectation that using large amounts of commodity hardware would scale, but it did not. We did not anticipate the number of failure modes we experienced and our printing process was delayed by approximately 1 and a half days.

**Ballot Delivery** Mail-in (absentee) ballots were delivered to the City Clerk on 16 October. Early, in-person voting ballots were delivered on October 27 for early voting on October 28, and all other ballots a couple of days later on October 30.

*Absentee ballots* were identical to in-person voting ballots except they did not contain online verification numbers and voters were not given any instructions on checking confirmation numbers online. They were returned by mail in double envelopes and scanned with the early votes. Confirmation numbers for these ballots were, however, made available online after scanning, so that there was no distinction in published data between absentee and in-person voted ballots.

The board decided to issue ballots without confirmation numbers due to the small number of anticipated absentee votes and the costs associated with mailing ballots with special pens. Mailing the ballots with confirmation codes would allow verification of confirmation codes, but opens up new attacks: the possibility of false charges of election fraud by adversaries who might expose confirmation codes and reprint ballots, or use expensive equipment to attempt to determine the invisible codes. Strong verification for absentee ballots is an ongoing research subject within the Scantegrity team.

*Early in-person voters* used Scantegrity ballots with all Scantegrity functionality, except that the early votes were scanned in after the polls closed on Election Day, and not by voters themselves. Voters were, however, provided verification cards and could check confirmation codes for these ballots online.

**Poll Worker Training** Several training sessions were held in the weeks prior to the election. Manuals from the previous election were updated and a companion guide was created with Scantegrity-specific instructions. Election judges were given these two manuals, and a member from our team demonstrated the voting process at one session.

[15] See `http://scantegrity.org/~carback1/ink` for more information on the printing process

**Voter Education** Voter education for this election focused on online verification. Articles in the City newspaper before the real election indicated that voters could check confirmation numbers online; this was also announced on the city's election website. [16]

**Scanner Setup** We attempted to minimize, not prevent, [17] the potential for using the wrong software by installing our software on top of Ubuntu Linux on SD flash cards, setting the "read-only" switch on each card, and setting up the software to read and write to USB sticks. We fingerprinted the first card after testing with the sha1sum utility and cloned it to a second card for the other netbook. Each netbook was set to boot from the card and BIOS configuration was locked with a password.

Both flash cards were checked with the sha1sum utility then placed into the netbook which was placed into a lock box and delivered to Takoma Park. The USB sticks were initialized with scanner configuration files. We uniquely identified each scanner by changing the ScannerID field in the configuration files, then we placed the corresponding USB sticks (3 for each netbook) into the lock box.

Upon delivery of the scanners the day before the election, we gave election officials the lock box keys and showed them how to open the lock boxes. We confirmed with election officials the contents of each box and the officials verified, with our assistance, that the USB memory sticks did not contain any ballot data by looking at the configuration file and making sure the ballot data file was blank. [18] To protect against virus infection on the sticks we set them to read-only for this procedure.

## 6.2 Election Day

On Election Day, November 3, 2009, polls were open from 7 am to 8 pm at a single polling location, the Takoma Park Community Center. Several members of the SVST were present through most of the day in the building in case of technical difficulty. One SVST member was permitted in the polling room at most times as an observer, and a couple of SVST members were present in the vestibule giving out and collecting survey forms through most of the day. Lillie Coney of the Electronic Privacy Information Center, who performed a print audit on the request of the Board of Elections, was present in the polling room through a large part of the day.

[16] `http://www.takomaparkmd.gov/clerk/election/2009/`

[17] Scantegrity would detect manipulation at the scanner. A better solution would use trusted hardware technology (e.g. a TPM [14]).

[18] These were the only 2 files on the disk at this time. Additionally, election officials did not check fingerprints on the flash cards. Since no 3rd party had reviewed the code or fingerprinted it they relied on our chain of custody.



**Starting the Election** The scanner was the only SVST equipment to set up and it was a turn key system. Election judges needed to plug in the USB sticks and power on the netbooks. The scanner was attached to a scanning apparatus, and cables were run into the lockbox that contained the netbook. When ready, the scanner would beep 3 times. After reading a ballot, the scanner would beep 1 time. During shutdown, the scanner would beep another 3 times. If there were any failure modes the scanner would beep continuously or not beep at all.

Election judges set up the check-in tables, pollbooks, and voting booths. The election started on time.

**Voting** The election proceeded quite smoothly, with very few (small) glitches. An SVST member was able to assist polling officials in fixing a problem with their poll books (not provided by Scantegrity). Voters had some initial problems with the use of the scanner and the privacy sleeve, some seeking assistance from election judges who also had difficulty. After an explanation to the election judges by the Chair of the Board of Elections, the use of the scanner was considerably smoother. With a few ballots, the privacy sleeve was not letting go of the ballots; one ballot was mangled considerably but scanned fine. Seventeen scanned ballots had lines on them that caused the scanner to be unable to read votes, and one ballot had alignment marks manipulated such that it was also unreadable. Images of all unreadable scans are saved, so we were able to manually enter in these votes. Of the seventeen ballots, many ballots had a line in the same location, which is consistent with there being a foreign substance on a ballot put into the scanner. These problems did not affect our ability to count the votes.

During the day, Ms. Coney chose about fifty ballots at random, uniformly distributed across wards, and exposed the confirmation codes for all options for the ballots. A copy of each ballot was made for her to take with her; the copies were signed by the Chair of the BoE. Neither Ms. Coney nor SVST members had any interaction with voters.

Towards the end of the day, after the local NPR station carried clips from an interview with the Chair of the Board of Elections and a voter, the polling station saw a large increase in the number of voters, with the line taking up much of the floor outside the polling room. The SVST prepared to print more ballots, but this was not required. The number of printed ballots ended up being almost twice the number of voted ballots.

Absentee and early voted ballots were scanned in after the closing of polls. Afterward, the scanners were shut down. The chief judge opened each lock box, set all sticks to read only, removed 2 USB sticks (leaving the third with the scanning netbook), and locked the lock box. Our team was given 1 stick for the ERM system. The other was kept by the city.

In *Meeting 3a*, trustees used Scantegrity code to generate results without provisional ballots at about 10 pm. The Chair of the Board of Election announced the results to those present at the polling place at the time (including candidates, their representatives, voters, etc.); this was also televised live by the local TV station. Confirmation codes and the election day tally were posted on the Scantegrity website.

## 6.3 After the Election

On the next day, around 2 pm, results including verified provisional ballots were published. Takoma Park representatives had announced a tally without provisional ballots the night before, and followed with the tally that included verified provisionals in accordance with standard Takoma Park procedures. The final *Meeting 3* results were published on November 4th just before midnight.

The number of registered voters were 10,934 and 1728 votes were cast (15.8%). The city-certified final tally for each contest is provided in Table 1. In each race, a majority was won after tallying after the voter's first choice.

**Hand Count and Certification** Following a hand count performed by representatives from both the SVST and Takoma Park, the Chair of the Board of Elections certified the results of the hand count to the City Council at 7 pm on November 5. The hand count and the Scantegrity count differed because officials were able to better determine voter intent during the hand count. For example, in the mayoral race, the scanner count determined that 646 votes were cast for candidate Schlegel, 972 for Williams, 15 for various write-in candidates, and 90 were not cast. The certified hand count totals were 664 votes for Schlegel, 1000 for Williams and 17 for write-in candidates. Thus 48 of a total of 1681 votes in this race would not have been counted by a scanner count alone. The discrepancy was caused by voters marking ballots outside of the designated marking areas. Such marks, while not read by the scanner by definition, are considered valid votes by Takoma Park law. Similarly, 8 of a total of 447 votes for Ward 1 council member, 8 of 251 for Ward 2, 16 of 431 for Ward 3, 10 of 210 for Ward 4, 2 of 81 for Ward 5 and 11 of 199 for Ward 6 were added to scanner vote totals after hand counting.

**Post-Election Audit** During *Meeting 4*, held on November 6 at 6 p.m., trustees used Scantegrity-written code to reveal all codes on voted ballots, and to reveal everything for all the ballots that were not spoiled or voted

| Mayor | Votes | Ward | Councilor | Votes | Ward | Councilor | Votes |
|---|---|---|---|---|---|---|---|
| Roger B. Schlegel | 664 | Ward 1 | Josh Wright | 434 | Ward 4 | Terry Seamens | 196 |
| Bruce Williams | 1000 | | Write-ins | 13 | | Eric Mendoza | 12 |
| Write-ins | 17 | Ward 2 | Colleen Clay | 236 | | Write-ins | 2 |
| | | | Write-ins | 15 | Ward 5 | Reuben Snipper | 71 |
| | | Ward 3 | Dan Robinson | 397 | | Write-ins | 10 |
| | | | Write-ins | 34 | Ward 6 | Navid Nasr | 61 |
| | | | | | | Fred Schultz | 138 |
| | | | | | | Write-ins | 0 |

Table 1: City certified election results for the Mayor's race and each City Councilman's race.

upon. Trustees also responded to pseudo-random challenges generated by stock market results at closing on November 6. Scantegrity published all data on November 7th around 9am. While the SVST could have chosen to use closing data on an earlier date, such as November 4 or November 5, which could have been more stable, the team chose to stick to its earlier-announced plan (of using the freshest stock market data) for the sake of consistency.

On November 9, 2009, Dr. Adida and Dr. Zagórski independently confirmed that Scantegrity correctly responded to all digital challenges. In particular, they confirmed that the tally computation audit data was correct. Both made available independently-written code on their websites that voters and others could use to check the tally computation commitments. The Chair of the BoE mentions that several voters have shown an interest in running the code made available by Drs. Adida and Zagórski, and that she expects that Takoma Park voters will use the code to perform some audits themselves in the next few months.

**Confirmation Codes and Complaints** The period for complaints regarding the election (including complaints about missing confirmation codes) expired at 6 pm on November 6. The Scantegrity website recorded 81 unique ballot ID verifications, of which about 66 (almost 4% of the total votes) were performed before the deadline. The SVST was also told by a BoE member that at least a few voters checked codes on auditor websites. Both Dr. Adida and Dr. Zagórski made the confirmation codes available on their websites after the election.

The number of voters who checked their ballots online before the Takoma Park complaint deadline (66), while not large, was sufficient to have detected (with high probability) any errors or fraud large enough to have changed the election outcome. (Detailed calculations omitted here; these calculations are not so simple, due to the use of IRV.)

Scantegrity received a single complaint by a voter who had trouble deciphering a digit in the code and noted it as "0," while the Scantegrity website presented it as "8." The voter requested that codes be printed more clearly in the future. He also stated that if he were not a trusting individual, he would believe that he had proof that his vote was altered.

All codes for all voted ballots were revealed after the dispute resolution period, and all commitments verified by two independent auditors, Dr. Adida and Dr. Zagórski. Hence, the probability that the code was in error is very small, albeit non-zero. Scantegrity does not believe the code was in error, and there were no other complaints regarding confirmation numbers.

**Print Audits** Dr. Zagórski provided an interface allowing Ms. Coney to check the commitments opened by Scantegrity in Meeting 4 against the candidate/confirmation-code correspondence on the ballots she audited. In her report [12], she confirmed that the correspondence between confirmation numbers and candidates on all the printed ballots audited by her was correctly provided by the interface.

**Followup** The Board of Elections and an SVST representative met to discuss the election and opportunities for improvement several weeks after the election. Both sides were largely satisfied with the election. Conversations have begun regarding the use of Scantegrity in the next municipal election at Takoma Park, to be held in November 2011. No decisions have been taken.

## 7 Surveys and Observations of Voter Experiences

To understand the experiences of voters and poll workers, we timed some of the voters as they voted, asked voters and poll workers to fill out two questionnaires, and informally solicited comments from voters as they left the precinct building. Approved by the Board of Elections and UMBC's Institutional Review Board, our procedures respected the constraint of not interfering with the election process. This section summarizes the results of our observations and surveys.

**Timing Data** Sitting unobtrusively as official observers in a designated area of the polling room for part of the day, two helpers (not members of the Scantegrity team) timed 93 voters as they carried out the voting process. Using stopwatches, they measured the number of seconds that transpired from the time the voter received a ballot to the time the voter began walking away from the scanner.

Voting times ranged from 55 secs. to 10mins. (the second longest time was 385 secs.), with a mean of 167 secs. and a median of 150 secs. On average, voters who appeared older took longer than voters who appeared younger. Most of the time was spent marking the ballot. The average time to vote was significantly faster than during the April 2009 mock election, when voters took approximately 8 mins. on average due primarily to scanning delays [26].

The observers noted that many voters did not fully use the privacy sleeve as intended, removing the ballot before scanning rather than inserting the privacy sleeve with the ballot into the scanning slot. Two of the 93 observed voters initially inserted the privacy sleeve upside-down, causing the ballot not to be fed into the scanner (even though the scanner could read the ballot in any orientation). A few ran into difficulties trying to insert the sleeve with one hand while holding something else in the other hand.

**Election Day Comments From Voters** As voters left the precinct building, members of the Scantegrity team conducting the written surveys, and a helper (a usability expert who is not a member of the Scantegrity team) solicited comments from voters with questions like, "What did you think of the new voting system?" The helper solicited comments 1:30-3:00pm and 7-8pm. A common response was, "It was easy."

Quite a few voters did not understand that they could verify their votes on-line and that, to do so, they had to write down the codenumbers revealed by their ballot choices. Some explained that they intentionally did not read any instructions because they "knew how to vote." Others failed to notice or understand instructions on posters along the waiting line, in the voting booth, on the ballot, and in the Takoma Park Newsletter.

In response, later in the day, we announced to voters as they entered the building that there is a new system; to verify your vote, write down the codenumbers. These verbal announcements seemed to have some positive effect, and there were fewer voter comments expressing lack of awareness of the verification option after we began the announcements. Nevertheless, some voters still were unaware of the verification option. It was a humbling experience to see first-hand how difficult it can be to get across the most basic points effectively, especially the first time a new system is used.

Some of the voters complained about the double-ended pen, not knowing which end to use, or having trouble writing in candidates with the chisel-point (the narrow point was intended for write-ins). A small number of voters had difficulty seeing the codenumbers, perhaps largely because repeatedly pressing too hard could erode the paper. A few voters expressed concern about the difficulty of writing down the codenumbers, had the ballot been much longer or had there been a large number of competing candidates.

Many voters expressed a strong confidence in the integrity of elections, while a small minority expressed sharp distrust in previous electronic election technology. These feelings seemed to be based more on a general subjective belief rather than on detailed knowledge of election procedures and technology. Similarly, those expressing strong confidence in Scantegrity seemed to like the concept of verification but did not understand in detail why Scantegrity provides high outcome assurance.

**Survey of Voter Experiences** As voters were leaving the precinct, we invited them to fill out two one-sided survey forms: a field-study questionnaire, and a demographics questionnaire. The field study asked voters about the voting system they just used, with most answers expressed on a seven-point Likert scale. The last question invited voters to make any additional suggestions or comments. Each pair of forms had matching serial numbers to permit correlation of the field study responses with demographics. 271 voters filled out the forms.

Fifty-one voters wrote comments on the questionnaires, often pointing out confusion about various aspects of the process but with no consistent theme. (1) Some were unaware of verification option. (2) Some did not realize they were supposed to write down codenumbers. (3) Some found the pens confusing to use: they did not realize that the pens would expose codenumbers, and they did not know which end to use. (4) Some found codenumbers were hard to read. (5) Some did not understand how to mark an IRV ballot. (6) Some did not know how to place the ballot into the scanner. (7) One had no difficulty but wondered if seniors or people who speak neither English nor Spanish might have difficulties. (8) One wondered if the government might be able to discern his vote by linking his IP address used during verification with his ballot serial number and noting the time that he was issued a ballot (this may be possible if the cryptography is broken or in other scenarios, but it would be more direct to have the scanner log how he voted). (9) Many

suggested that it would have been helpful to have better instructions, including instruction while they wait in line.

Figure 3 shows how voters responded to four questions from the field study questionnaire. These results strongly show that voters found the voting system easy to use (Question 5), and that they had confidence in the system (Question 13). Question 10 showed that the option to check votes on line increased voter confidence in the election results. Question 9 showed that voters had confidence that the receipt alone did not reveal how they voted; this finding is notable given that it is widely suspected that many people erroneously believe that all E2E receipts reveal ballot choices. We plan to present detailed analysis of our complete survey data in a separate companion paper.

**Survey of Poll Worker Experiences** Each of the twelve poll workers was given an addressed and stamped envelope with two questionnaires (field study and demographics) to fill out and mail to the researchers after the election. The field study focused on their experiences administering Scantegrity, with most answer expressed on a seven-point Likert scale. This questionnaire also included four open-ended questions. Each pair of forms had matching serial numbers. Five forms were returned.

Poll workers noted the following difficulties. (1) There was too much information. (2) Some voters did not understand what to do, including how to create a receipt. (3) Some voters did not understand how to mark an IRV ballot. (4) The privacy sleeve was hard to use with one hand. (5) The double-ended pens created confusion. (6) Voters, poll workers, and the Scantegrity team have different needs. One wondered if Scantegrity was worth the extra trouble.

They offered the following suggestions: (1) Simplify the ballot. (2) Provide receipts so that voters do not have to copy codenumbers. (3) Develop better pre-election voter education.

## 8 Discussion and Lessons Learned

Overall, this project should be deemed a success: the goals of the election were met, and there were no major snafus. Many aspects of the Scantegrity design and implementation worked well, while some could be improved in future elections.

**Technology Challenges** Perhaps the most challenging aspect for future elections is scaling up ballot printing. The printers we used were not very reliable.

Variations on the Scantegrity design worth exploring include the printing of voter receipts (rather than having voters copy confirmation codes by hand)—there are clearly security aspects to handle if one does this. The design should also be extended for better accessibility. The special pen might be improved by having only a single medium-tip point, rather than two tips of different sizes. The scanning operation and its interaction with the privacy sleeve should be studied and improved.

The website, while sufficient, might utilize existing research in distributed systems to reduce the expectations on observers and voters. The scanner could also be improved with more sophisticated image analysis, and also to better handle unreadable ballots. It only occured to us after the election that the write-in resolution process could have greater utility if it were expanded to deal with unreadable and unclear ballots.

**Real World Deployment of Research Systems** As is common with many projects, too much was left until the last minute. Better project management would have been helpful, and key aspects should have been finalized earlier. Materials and procedures should be more extensively tested beforehand.

One of the most important lessons learned is the value of close collaboration and clear communication between election officials and the election system providers (whether they be researchers or vendors).

Another lesson learned is that it is both important to provide voters with clear explanations of the new features of a voting system, and to do so efficiently, with minimal impact on throughput. Resolving the tension between these requirements definitely needs further exploration. For example, it might be worthwhile to have an instructional video explaining the Scantegrity system that voters could watch as they come in. The permanent adoption of Scantegrity II in a jurisdiction would, however, alleviate the educational burden over time, as voters learn the system's features in successive elections.

**Comparison with post-election audits** It is interesting to compare Scantegrity with the other major technique for election outcome verification: post-election audits. Because these audits do not allow anyone to check that a particular ballot was counted correctly, they do not provide the level of integrity guarantee provided by Scantegrity.

Post-election audits, even those with redundant digital and physical records like optical scan systems, only address errors or malfeasance in the counting of votes and not in the chain of custody. [19] In contrast, end-to-end

[19] Having multiple records may make an attacker's job harder, but note that the attacker only has to change the record that will ultimately be used and/or trusted (not necessarily both). Also, redudancy can work against a system, as changing a digital record in an obviously malicious way may allow time for a more subtle manipulation of the physical record.



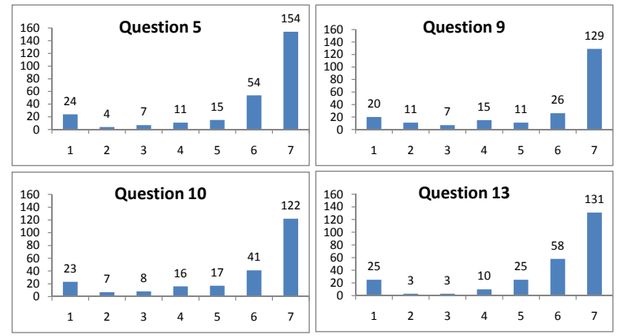



Figure 3: Voter responses to Survey Questions 5, 9, 10, 13 from all 271 voters completing the survey. Using a seven-point Likert scale, voters indicated how strongly they agreed or disagreed with each statement about the voting system they had just used (1 = strongly disagree, 7 = strongly agree). Each histogram shows the number of voters responding for each of the seven agreement levels. The four questions shown are the following: (5) Overall, the voting system was easy to use. (9) I have confidence that my receipt by itself does not reveal how I voted. (10) The option to verify my vote online afterwards increases my confidence in the election results. (13) I have confidence in this voting system.

voting systems such as Scantegrity provide a "verifiable chain of custody." Voters can check that their ballots are included in the tally, and anyone—not just a privileged group of auditors—can check that those ballots are tallied as intended.

It must be admitted, however, that the additional integrity benefits provided by Scantegrity II come at the cost of somewhat increased complexity and at the cost of an increased (but manageable) risk to voter privacy (since ballots are uniquely identifiable). That said, some jurisdictions and/or election systems require or use serial numbers on ballots anyway, and we have proposed several possible approaches to appropriately destroy or obfuscate serial number information. Furthermore, it can be argued that a voter wishing to "fingerprint" a ballot can do so without being detected in current paper ballot systems simply by marking ovals in distinctive ways.

## 9 Conclusions

Traditional opscan voting systems have the clear benefit that "votes are verifiably cast as intended"—the voter can see for herself that the ballot is correctly filled out. Yet once her ballot is cast, the voter must place her trust in others that ballots are safely collected and correctly counted. With end-to-end voting systems these last two operations (collecting ballots and counting them) are verifiable as well: voters can verify—using their receipt and a website—that their ballot is safely collected with the others, and anyone can use the website data to verify that the ballots have been correctly counted. The Scantegrity II voting system provides such end-to-end verification capability as an overlay on top of traditional opscan technology. Further development should improve scalability (esp. printing), usability (e.g. with printed receipts) and accessibility of the Scantegrity II system.

The successful use of the Scantegrity II voting system in the Takoma Park election of November 3, 2009 demonstrates that voters and election officials can use sophisticated cryptographic techniques to organize a transparent secret ballot election with a familiar voting experience. The election results show considerable satisfaction by both voters and pollworkers, indicating that end-to-end voting technology has matured to the point of being ready and usable for real binding governmental elections. This paper thus documents a significant step forward in the security and integrity of voting systems as used in practice.

**Acknowledgments** The authors would like to acknowledge the contributions of the voters of Takoma Park, the City Clerk, the Assistant City Clerk, all Board of Elections members since 2008 when this project was first proposed, and the independent auditors—Lillie Coney, Ben Adida and Filip Zagórski—to the success of the election. Vivek Relan and Bhushan Sonawane timed voters as they voted and helped assemble the privacy sleeves. Lynn Baumeister interviewed some voters as they left the precinct. Cory Jones provided general assistance and Alex Florescu and Jan Rubio assisted with ink creation.

Alan T. Sherman was supported in part by the Depart-

ment of Defense under IASP grants H98230-08-1-0334 and H98230-09-1-0404. Poorvi L. Vora was supported in part by The National Science Foundation under grant CNS 0831149. Jeremy Clark and Aleksander Essex were supported in part by Natural Sciences and Engineering Research Council of Canada (NSERC).

**Disclosure** Portions of the Scantegrity system may be covered by pending patents under applications US 2008/0272194, and US 2009/0308922. All source code was released under the GPLv2 software license. [20]

## References

[1] ADIDA, B. *Advances in Cryptographic Voting Systems*. PhD thesis, MIT EECS Dept., 2006.

[2] ADIDA, B. Helios: web-based open-audit voting. In *Proceedings of the 17th USENIX Security Symposium* (2008), pp. 335–348.

[3] ADIDA, B., DEMARNEFFE, O., PEREIRA, O., AND QUISQUATER, J.-J. Electing a University President using Open-Audit Voting: Analysis of real-world use of Helios. *Proceedings of the Electronic Voting Technology Workshop / Workshop on Trustworthy Elections* (August 2009).

[4] BENALOH, J. Simple verifiable elections. In *Proceedings of the 2006 USENIX/ACCURATE Electronic Voting Technology Workshop* (Berkeley, CA, USA, 2006), USENIX Association.

[5] CHAUM, D. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (1981), 84–90.

[6] CHAUM, D. Secret-Ballot Receipts: True Voter-Verifiable Elections. *IEEE Security and Privacy 2*, 1 (2004), 38–47.

[7] CHAUM, D., CARBACK, R., CLARK, J., ESSEX, A., POPOVENIUC, S., RIVEST, R. L., RYAN, P. Y. A., SHEN, E., AND SHERMAN, A. T. Scantegrity II: end-to-end verifiability for optical scan election systems using invisible ink confirmation codes. In *Proceedings of the 2008 USENIX/ACCURATE Electronic Voting Technology Workshop* (2008), pp. 1–13.

[8] CHAUM, D., CARBACK, R. T., CLARK, J., ESSEX, A., POPOVENIUC, S., RIVEST, R. L., RYAN, P. Y. A., SHEN, E., SHERMAN, A. T., AND VORA, P. L. Scantegrity II: End-to-End Verifiability by Voters of Optical Scan Elections Through Confirmation Codes. *IEEE Trans. on Information Forensics and Security, special issue on electronic voting 4*, 4 (Dec. 2009), 611–627.

[9] CHAUM, D., ESSEX, A., CARBACK, R., CLARK, J., POPOVENIUC, S., SHERMAN, A. T., AND VORA, P. Scantegrity: End-to-End Voter Verifiable Optical-Scan Voting. *IEEE Security and Privacy Magazine 6*, 3 (May/June 2008), 40–46.

[10] CHAUM, D., RYAN, P. Y., AND SCHNEIDER, S. A. A practical, voter-verifiable, election scheme. Technical Report Series CS-TR-880, University of Newcastle Upon Tyne, December 2004.

[11] CLARK, J., ESSEX, A., AND ADAMS, C. Secure and observable auditing of electronic voting systems using stock indices. In *Proceedings of the 2007 IEEE Canadian Conference on Electrical and Computer Engineering* (2007).

[12] CONEY, L. Report on the Manual Ballot Audit: Takoma Park, Maryland, November 3 2009 Election, 19 November 2009. Electronic Privacy Information Center, `http://epic.org/privacy/voting/takoma_park_audit.pdf`.

[13] ESSEX, A., CLARK, J., CARBACK, R. T., AND POPOVENIUC, S. Punchscan in Practice: an E2E Election Case Study. In *Proceedings of the 2007 IAVoSS Workshop on Trustworthy Elections* (2007).

[14] FINK, R. A., SHERMAN, A. T., AND CARBACK, R. Tpm meets DRE: reducing the trust base for electronic voting using trusted platform modules. *Trans. Info. For. Sec. 4*, 4 (2009), 628–637.

[15] FISHER, K., CARBACK, R., AND SHERMAN, A. T. Punchscan: Introduction and System Definition of a High-Integrity Election System. In *Proceedings of the 2006 IAVoSS Workshop on Trustworthy Elections* (2006).

[16] HOSP, B., JANSON, N., MOORE, P., ROWE, J., SIMHA, R., STANTON, J., AND VORA, P. Citizen-Verified Voting. Presentation at DIMACS Workshop on Electronic Voting – Theory and Practice, May 2004, `http://dimacs.rutgers.edu/Workshops/Voting/slides/vora.ppt`.

[17] HUBBERS, E., JACOBS, B., SCHOENMAKERS, B., VAN TILBORG, H., AND DE WEGE, B. Description and Analysis of RIES, June 2008.

[18] KUTYLOWSKI, M., AND ZAGÓRSKI, F. Verifiable Internet Voting Solving Secure Platform Problem. In *Advances in Information and Computer Security, Lecture Notes in Computer Science* (2007), vol. 4752, pp. 199–213.

[19] NEFF, C. A. Practical high certainty intent verification for encrypted votes, 2004.

[20] OFFICE FOR DEMOCRATIC INSTITUTIONS AND HUMAN RIGHTS. The Netherlands Parliamentary Elections 22 November 2006 OSCE/ODIHR Election Assessment Mission Report, March 12 2007. 28 pages.

[21] POPOVENIUC, S., AND HOSP, B. An introduction to punchscan. In *Proceedings of the 2006 IAVoSS Workshop on Trustworthy Elections* (2006).

[22] POPOVENIUC, S., AND STANTON, J. Undervote and Pattern Voting: Vulnerability and a mitigation technique. In *IAVoSS Workshop On Trustworthy Elections (WOTE 2007)* (University of Ottawa, Ottawa, Canada, June 2007).

[23] POPOVENIUC, S., AND VORA, P. L. A framework for secure electronic voting. In *Proceedings of the 2008 IAVoSS Workshop on Trustworthy Elections* (2008).

[24] SANDLER, D. R., DERR, K., AND WALLACH, D. S. VoteBox: a tamper-evident, verifiable electronic voting system. In *Proceedings of the 17th USENIX Security Symposium* (2008).

[25] SHAMIR, A. How to Share a Secret. *CACM 22*, 11 (Nov 1979), 612–613.

[26] SHERMAN, A. T., CHAUM, D., CLARK, J., ESSEX, A., HERRNSON, P. S., MAYBERRY, T., POPOVENIUC, S., RIVEST, R. L., SHEN, E., SHERMAN, A. T., AND VORA, P. L. Scantegrity Mock Election at Takoma Park. In *Proceedings of the 4th International Conference on Electronic Voting (EVOTE 2010)* (2010).

[27] City of Takoma Park, Maryland City Election November 3, 2009 Certification of Election Results, November 2009. `http://www.takomaparkmd.gov/clerk/election/2009/results/2009cert.pdf`.

[28] VoComp Voting System Competition. July, 2007. Portland, Oregon. `http://www.vocomp.org`.

[20] `http://www.gnu.org/licenses/gpl-2.0.html`



# Acoustic Side-Channel Attacks on Printers

*Michael Backes*[1,2], *Markus Dürmuth*[1], *Sebastian Gerling*[1], *Manfred Pinkal*[3], *Caroline Sporleder*[3]

[1] *Saarland University, Computer Science Department, Saarbrücken, Germany*

[2] *Max Planck Institute for Software Systems (MPI-SWS)*

[3] *Saarland University, Computer Linguistics Department, Saarbrücken, Germany*

## Abstract

We examine the problem of acoustic emanations of printers. We present a novel attack that recovers what a dot-matrix printer processing English text is printing based on a record of the sound it makes, if the microphone is close enough to the printer. In our experiments, the attack recovers up to 72 % of printed words, and up to 95 % if we assume contextual knowledge about the text, with a microphone at a distance of 10cm from the printer. After an upfront training phase, the attack is fully automated and uses a combination of machine learning, audio processing, and speech recognition techniques, including spectrum features, Hidden Markov Models and linear classification; moreover, it allows for feedback-based incremental learning. We evaluate the effectiveness of countermeasures, and we describe how we successfully mounted the attack in-field (with appropriate privacy protections) in a doctor's practice to recover the content of medical prescriptions.

## 1 Introduction

Information leakage caused by emanations from electronic devices has been a topic of concern for a long time. The first publicly known attack of this type, published in 1985, reconstructed the monitor's content from its electromagnetic emanation [36]. The military had prior knowledge of similar techniques [41, 20]. Related techniques captured the monitor's content from the emanations of the cable connecting the monitor and the computer [21], and acoustic emanations of keyboards were exploited to reveal the pressed key [3, 42, 7]. In this work we examine the problem of acoustic emanations of dot-matrix printers.

**Dot matrix printers? Didn't these printers vanish in the 80s already?** Although indeed outdated for private use, dot-matrix printers continue to play a surprisingly prominent role in businesses where confidential information is processed. We commissioned a representative survey from a professional survey institute [26] in Germany on this topic, with the following major lessons learned (Figure 1 contains additional information from this survey):

- About 60 % of all doctors in Germany use dot matrix printers, for printing the patients' health records, medical prescriptions, etc. This corresponds to about 190,000 doctors and an average number of more than 2.4 million records and prescriptions printed on average per day.
- About 30 % of all banks in Germany use dot matrix printers, for printing account statements, transcripts of transactions, etc. This corresponds to 14,000 bank branches and more than 1.2 million such documents printed on average per day.
- Only about 5 % of these doctors and about 8 % of these banks currently plan to replace dot matrix printers. The reasons for the continued use of dot-matrix printers are manifold: robustness, cheap deployment, incompatibility of modern printers with old hardware, and overall the lack of a compelling business reason of IT laymen why working IT hardware should be modernized.
- Several European countries (e.g., Germany, Switzerland, Austria, etc.) require by law the use of dot-matrix (carbon-copy) printers for printing prescriptions of narcotic substances [8].

### 1.1 Our contributions

We show that printed English text can be successfully reconstructed from a previously taken recording of the sound emitted by the printer. The fundamental reason why the reconstruction of the printed text works is that, intuitively, the emitted sound becomes louder if more needles strike the paper at a given time (see Figure 2 for

| DOCTORS ($n$=541 ASKED) | |
|---|---|
| Use dot-matrix printers | 58.4 % |
| - for general prescriptions | 79.4 % |
| - for other usages | 84.5 % |
| Printer placed in proximity of patients | 72.2 % |
| Replacement planned | 4.7 % |

| BANKS ($n$=524 ASKED) | |
|---|---|
| Use dot-matrix printers | 30.0 % |
| - for bank statement printers | 29.9 % |
| - for other usages | 83.4 % |
| Printer placed in proximity of customers | 83.4 % |
| Replacement planned | 8.3 % |

Figure 1: Main results of the survey on the usage of dot-matrix printers in doctor's practices and banks [26]. Other printer usages reported in the survey comprise: "certificate of incapacity for work, transferal to another doctor, hospitalization, and receipts" for doctors, and "account book, PIN numbers for online banking, supporting documents, ATMs" for banks.

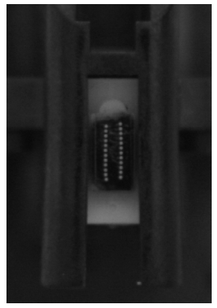


Figure 2: Print-head of an Epson LQ-300+II dot-matrix printer, showing the two rows of needles.

a typical setting of 24 needles at the printhead). We verified this intuition and we found that there is a correlation between the number of needles and the intensity of the acoustic emanation (see Figure 3). We first conduct a training phase where words from a dictionary are printed, and characteristic sound features of these words are extracted and stored in a database. We then use the trained characteristic features to recognize the printed English text. (Training and recognition on a letter basis, similar to [42], seems more appealing at first glance since it naturally comprises the whole vocabulary. However, the emitted sound is strongly blurred across adjacent letters, rendering a letter-based approach much poorer than the word-based approach, even if spell-checking is used, see below).

This task is not trivial. Major challenges include: (i) Identifying and extracting sound features that suitably capture the acoustic emanation of dot-matrix printers; (ii) Compensating for the blurred and overlapping features that are induced by the substantial decay time of the emanations; (iii) Identifying and eliminating wrongly recognized words to increase the overall percentage of correctly identified words (recognition rate).

**Overview of the approach.** Our work addresses these challenges, using a combination of machine learning techniques for audio processing and higher-level information about document coherence. Similar techniques are used in language technology applications, in particular in automatic speech recognition.

First, we develop a novel feature design that borrows from commonly used techniques for feature extraction in speech recognition and music processing. These techniques are geared towards the human ear, which is limited to approx. 20 kHz and whose sensitivity is logarithmic in the frequency; for printers, our experiments show that most interesting features occur above 20 kHz, and a logarithmic scale cannot be assumed. Our feature design reflects these observations by employing a sub-band decomposition that places emphasis on the high frequencies, and spreading filter frequencies linearly over the frequency range. We further add suitable smoothing to make the recognition robust against measurement variations and environmental noise.

Second, we deal with the decay time and the induced blurring by resorting to a word-based approach instead of decoding individual letters. A word-based approach requires additional upfront effort such as an extended training phase (as a word-based dictionary is larger), and it does not permit us to increase recognition rates by using, e.g., spell-checking. Recognition of words based on training the sound of individual letters (or pairs/triples of letters), however, is infeasible because the sound emitted by printers blurs too strongly over adjacent letters. (Even words that differ considerably on the letter basis may yield highly similar overall sound features, which complicates the subsequent post-processing, see below.) This complication was not present in earlier work on acoustic emanations of keyboards, since the time between two consecutive keystrokes is always large enough that blurring was not an issue [42].

Third, we employ speech recognition techniques to increase the recognition rate: we use Hidden Markov Models (HMMs) that rely on the statistical frequency of sequences of words in English text in order to rule out in-



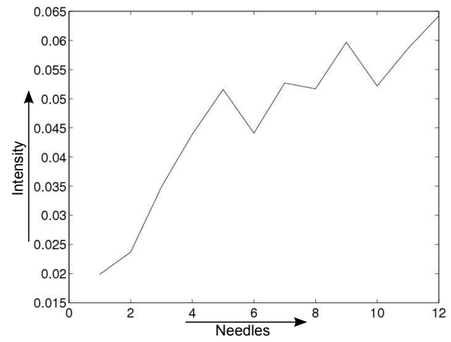



Figure 3: Graph showing the correlation between the number of needles striking the ribbon and the measured acoustic intensity.

correct word combinations. The presence of strong blurring, however, requires the use of at least 3-grams on the words of the dictionary to be effective, causing existing implementations for this task to fail because of memory exhaustion. To tame memory consumption, we implemented a delayed computation of the transition matrix that underlies HMMs, and in each step of the search procedure, we adaptively removed the words with only weakly matching features from the search space.

**Experiments, underlying assumptions and limitations.** Before we describe our experiments, let us be clear about the underlying assumptions that render our approach possible. (i) The microphone (or bug) has to be (surreptitiously) placed in close proximity (about 10cm) of the printer. (ii) Because our approach is word-based for the reasons described above, it will only identify words that have been previously trained; feedback-based incremental training of additional words is possible. While this is less a concern for, e.g., recovering general English text and medical prescriptions, it renders the attack currently infeasible against passwords or PIN numbers. In the bank scenario, the approach can still be used to identify, e.g., the sender, recipient, or subject of a transaction. (iii) Conducting the learning phase requires access to a dot matrix printer of the same model. There is no need to get hold of the actual printer at which the target text was printed. (iv) If HMM-based post-processing is used, a corpus of (suitable) text documents is required to build up the underlying language model. Such post-processing is not always necessary, e.g., our in-field attack in a doctor's practice described below did not exploit HMMs to recover medical prescriptions.

We have built a prototypical implementation that can bootstrap the recognition routine from a database of featured words that have been trained using supervised learning. We applied this implementation to four different English text documents, using a dictionary of about 1,400 words (including the 1,000 most frequently used English words and the words that additionally occur in these documents, see the second assumption above) and a general-purpose corpus extracted from stable Wikipedia articles that the HMM-based post-processing relies upon. The prototype automatically recognizes these texts with recognition rates of up to 72 %. To investigate the impact of HMM-based post-processing with a domain-specific corpus instead of a general-purpose corpus on the recognition rate, we considered two additional documents from a privacy-sensitive domain: living-will declarations. We used publicly available living-will declarations to extract a specialized corpus, thereby also increasing the dictionary to 2,150 words. Our prototype automatically recognized the two target declarations with recognition rates of about 64 % using the general-purpose corpus, and increased the recognition rates to 72 % and 95 %, respectively, using the domain-specific corpus. This shows that, somewhat expectedly, HMM-based post-processing is particularly worthwhile if prior knowledge about the domain of the target document can be assumed.

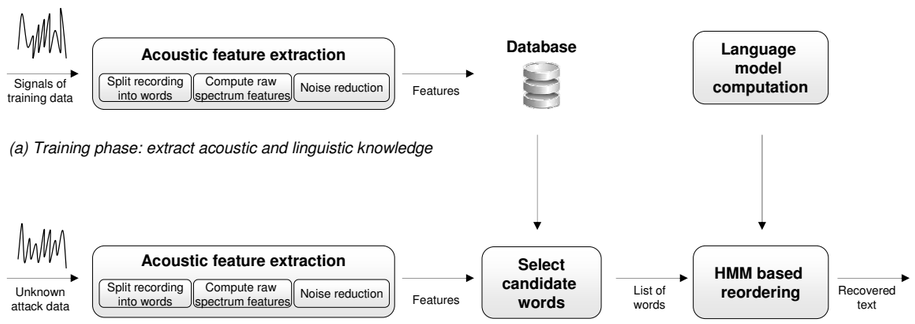
We have identified and evaluated countermeasures that prevent this kind of attack. We found that fairly simple countermeasures such as acoustic shielding and ensuring a greater distance between the microphone and the printer suffice for most practical purposes.

Furthermore, we have successfully mounted the attack in-field in a doctor's practice to recover the content of medical prescriptions. (For privacy reasons, we asked for permission upfront and let the secretary print fresh prescriptions of an artificial client.) The attack was observer-blind and conducted under realistic – and arguably even pessimistic – circumstances: during rush hour, with many people chatting in the waiting room.

## 1.2 Related work

Military organizations investigated compromising emanations for many years. Some of the results have been declassified: the Germans spied on the French field phone lines in World War I [6], the Japanese spied on American cipher machines using electromagnetic emanations in 1962 [1], the British spied on acoustic emanation of (mechanical) Hagelin encryption devices in the Egyptian embassy in 1956 [39, p. 82], and the British spied on parasitic signals leaked by the French encryption machines in the 1950s [39, p. 109f].

The first publicly known attack we are aware of was published in 1985, and exploited electromagnetic radiation of CRT monitors [36, 16]. Since then, various forms of emanations have been exploited. Electromag-




*(a) Training phase: extract acoustic and linguistic knowledge*

*(b) Recognition phase: recognize printed text using acoustic and linguistic features*

Figure 4: Overview of the attack.

netic emanations that constitute a security threat to computer equipment result from poorly shielded RS-232 serial lines [35], keyboards [2], as well as the digital cable connecting modern LCD monitors [21]. We refer to [22] for a discussion of the security limits for electromagnetic emanation. The time-varying diffuse reflections of the light emitted by a CRT monitor can be exploited to recover the original monitor image [19]; compromising reflections were studied in [5, 4]. Information leaking from status LEDs was studied in [25].

Acoustic emanations were shown to divulge text typed on ordinary keyboards [3, 42, 7], as well as information about the CPU state and the instructions that are executed [33]. Acoustic emanations of printers were briefly mentioned before [10]; it was solely demonstrated that the letters "W" and "J" can be distinguished. This study did not determine whether any other letters can be distinguished, let alone if a whole text can be reconstructed by inspection of the recording, or even in an automated manner.

Several techniques from audio processing are adapted for use in our system. A central technique is feature extraction. We use features based on sub-band decomposition [27]. Alternative feature designs are based on the (Short-time) Fast Fourier Transform [34], or on the Cepstrum transformation [11] which is the basis for Mel Frequency Cepstral Coefficients (MFCC) [23, 15, 9, 24, 30].

### 1.3 Paper outline

Section 2 presents a high-level description of our new attack, with full technical details given in Section 3. Section 4 presents experimental results. Section 5 describes the attack we conducted in-field. We conclude with some final remarks in Section 6.

## 2 Attack Overview

In this section, we survey our attack without delving into the technical details. We consider the scenario that English text containing potentially sensitive information is printed on a dot-matrix printer, and the emitted sound is recorded. We develop a methodology that on input the recording automatically reproduces the printed text. Figure 4 presents a holistic overview of the attack.

The first phase (Figure 4(a)) constitutes the *training* phase that can take place either before or after the attack. In this phase, a sequence of words from a dictionary is printed, and characteristic sound features of each word are extracted and stored in a database. For obtaining the best results, the setting should be close to the setting in which the actual attack is mounted, e.g., similar environmental noise and acoustics. Our experiments indicate that creating sufficiently good settings for reconstruction does not pose a problem, see Section 4.3.2. The main steps of the training phase are as follows:

1. *Feature extraction.* We use a novel feature design that borrows from commonly used techniques for feature extraction in speech recognition and music processing. In contrast to these areas, our experiments show that most interesting features for printed sounds occur above 20 kHz, and that a logarithmic scale cannot be assumed for them. We hence split the recording into single words based on the intensity of the frequency band between 20 kHz and 48 kHz, and spread the filter frequencies linearly over the frequency range. We subsequently use digital filter banks to perform sub-band decomposition on each word [27]. As discussed in Section 3.1, sub-band decomposition gives better results than simple FFT because of better time resolution. The output of sub-band decomposition is smoothed to make it more robust to measurement variations and environmental noise. The extracted features are stored in a database.

2. *Computation of language models.* To solve the recognition task, we will complement acoustic information with information about the occurrence likelihood of words in their linguistic context (e.g., the sequence "such as the" is much more likely than "such of the"). More specifically, we estimate for each word in our lexicon $n$-gram probabilities, i.e., the likelihood that the word occurs after a sequence of $n-1$ given words. These probabilities make up a *(statistical) language model*. Probabilities are computed based on frequency counts of $n$-place sequences ($n$-grams) from a corpus of text documents. We need to extract these frequencies from a sufficiently large corpus, which makes up the second step of the training phase. In our experiments, we used 3-gram frequencies extracted from a corpus of 10 million words of English text. For our domain-specific experiments, we used a corpus of living-will declarations consisting of 14,000 words of English text.

The second phase (Figure 4(b)), called the *recognition* phase, uses the characteristic features of the trained words to recognize new sound recordings of printed text, complemented by suitable language-correction techniques. The main steps are as follows:

1. *Select candidate words.* We start by extracting features of the recording of the printed target text, as in the first step of the training phase. Let us call the obtained sequence of features target features whereas the features from the training phase stored in the database are henceforth referred to as trained features. Now, we subsequently compare, on a word-by-word basis, the obtained target features with the trained features of the dictionary stored in the database.

   If the features extracted from different recordings of the same word were always identical, one would obtain a unique correspondence between trained features and target features (under the assumption that all text words are in the dictionary). However, measurement variations, environmental noise, etc. show that this is not the case. Multiple recordings of the same word sometimes yield different features; for example, printing the same word at different places in the document results in differing acoustic emanations (Figure 10 illustrates how a single vertical line already differs in the intensity); conversely, recordings of words that differ significantly in their spelling might yield almost identical sound features. We hence let the selected, trained word be a random variable conditioned on the printed word, i.e., every trained word will be a candidate with a certain probability. Using sufficiently good feature extraction and distance computations between two features, the probabilities of one or a few such trained words will dominate for each printed word. The output of the first recognition step is a list of most likely candidates, given the acoustic features of the target word.

2. *Language-based reordering to reduce word error rate.* We finally try to find the most likely sequence of printed words given a ranked list of candidate words for each printed word. Although always naively picking the most likely word based on the acoustic signal might already yield a suitable recognition quality, we employ Hidden Markov Model (HMM) technology, in particular language models and the Viterbi algorithm (see Section 3.3.3 for details), which is regularly used in speech recognition, to determine the most likely sequence of printed words. Intuitively, this technology works well for us because most errors that we encounter in the recognition phase are due to incorrectly recognized words that do not fit the context; by making use of linguistic knowledge about likely and unlikely sequences of words, we have a good chance of detecting and correcting such errors. The use of HMM technology yields accuracy rates of 70 % on average for words for the general-purpose corpus, and up to 95 % for the domain-specific corpus, see Section 3.3 for details.

   We modified the Viterbi algorithm to meet our specific needs: first, the standard algorithm accepts as input a sequence of outputs, while we get for each position an ordered list of likely candidates, and we want to profit from this extra knowledge; second, we need to decrease memory usage, since a standard implementation would consume more than 30 GB of memory.

## 3 Technical Details

In this section we provide technical details about our attack, including the background in audio processing and Hidden-Markov Models.

### 3.1 Feature extraction

We are faced with an audio file sampled at 96 kHz with 16bit.

To *split the recording into words*, we use a threshold on the intensity of the frequency band from 20 kHz to 48 kHz. For printers, our experiments have shown that most interesting features occur above 20 kHz, making this frequency range a reliable indicator despite its simplicity; ignoring the lower frequencies moreover avoids most noise added by the movement of the print-head etc.

From the split signal, we *compute the raw spectrum features* by sub-band decomposition, a common technique in different areas of audio processing. The signal is filtered by a filter bank, a parallel arrangement of several bandpass filters tuned in steps of 1 kHz over the range from 1 kHz to 48 kHz.

For *noise reduction* the output of the filters is smoothed, normalized, the amount of data is reduced (the maximal value out of 5 is kept), and smoothed again. The result is appropriately discretized over time and forms a set of vectors, one vector for each filter.

The feature design has a major influence on the running time and storage requirements of the subsequent audio processing. We have experimented with several alternative feature designs, but obtained the best results with the design described above. The (Short-time) Fast Fourier Transform (SFFT) [34] seems a natural alternative to sub-band decomposition. There is, however, a trade-off between the frequency and the time resolution, and we obtain worse results in our setting when we used SFFTs, similar to earlier observations [42].

## 3.2 Select candidate words

Deciding which database entry is the best match for a recording is based on the following distance function defined on features; the tool outputs the 30 most similar entries along with the calculated distance. Given the features extracted from the recording $(\vec{x}_1, \ldots, \vec{x}_t)$ and the features of a single database entry $(\vec{y}_1, \ldots, \vec{y}_t)$ we compute the angle between each pair of vectors $\vec{x}_i, \vec{y}_i$ and sum over all frequency bands:

$$\Delta((\vec{x}_1, \ldots, \vec{x}_t), (\vec{y}_1, \ldots, \vec{y}_t)) = \sum_{i=1,\ldots,t} \arccos\left(\frac{\vec{x}_i \cdot \vec{y}_i}{|\vec{x}_i| \cdot |\vec{y}_i|}\right).$$

To increase robustness and decrease computational complexity in practical scenarios, some problems need to be addressed: First, our implementation of cutting the audio file sometimes errs a bit, which leads to slightly non-matching samples. Thus we consider minor shiftings of each sample by tiny amounts (two steps in each direction, or a total of 5 measurements) and take the minimum angle (i.e., the maximum similarity). Second, for a similar reason, we tolerate some deviation in the length of the features. We punish too large deviations by multiplying with a factor of 1.2 if the length of the query and the database entry differ by more than a defined threshold. The factor and the threshold are derived from our experiments. Third, we discard entries whose length deviates from the target feature by more than 15 % in order to speed up the computation.

Using the angle to compare features is a common technique. Other approaches that are used in different scenarios include the following: Müller et al. present an audio matching method for chroma based features that handles tempo differences [28]. Logan and Salomon use signatures based on clustered MFCCs as input for the distance calculation in [24]. Furthermore, they use the earth mover's distance [32] for the signatures (minimum amount of work to transform one signature into another) and the Kullback Leibler (KL) distance for the clusters inside the signature as distance measures.

## 3.3 Post-processing using HMM technology

In this section we describe techniques based on language models to further improve the quality of reconstruction. These improve the word recognition rate from 63 % to 70 % on average, and up to 72 % in some cases. The domain-specific HMM-based post-processing even achieves recognition rates of up to 95 %.

### 3.3.1 Introduction to HMMs

Hidden Markov models (HMMs) are graphical models for recovering a sequence of random variables which cannot be observed directly from a sequence of (observed) output variables. The random variables are modeled as hidden states, the output variables as observed states. HMMs have been employed for many tasks that deal with natural language processing such as speech recognition [31, 18, 17], handwriting recognition [29] or part-of-speech tagging [12, 14].

Formally, an HMM of order $d$ is defined by a five-tuple $\langle Q, O, A, B, I \rangle$, where $Q = (q_1, q_2, ..., q_N)$ is the set of (hidden) states, $O = (o_1, o_2, ..., o_M)$ is the set of observations, $A = Q^{d+1}$ is the matrix of state transition probabilities (i.e., the probability to reach state $q_{d+1}$ when being in state $q_d$ with history $q_1, \ldots, q_{d-1}$), $B = Q \times O$ are the emission probabilities (i.e., the probability of observing a specific output $o_i$ when being in state $q_j$), and $I = Q^d$ is the set of initial probabilities (i.e., the probability of starting in state $q_i$). Figure 5 shows a graphical representation of an HMM, where unshaded circles represent hidden states and shaded circles represent observed states.



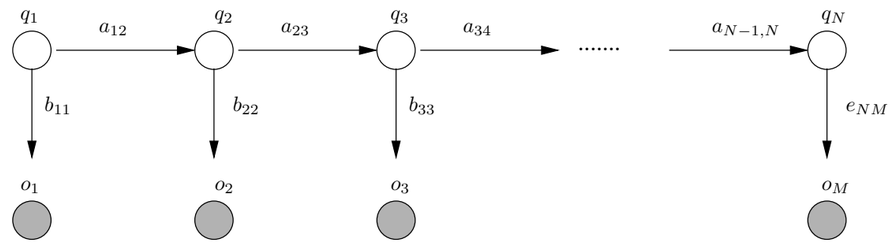



Figure 5: Hidden Markov Model

In our setting the words that were printed are unknown and correspond to the hidden states. The observed states are the output of the first stage of reconstruction from the acoustic signals emitted by the printer. What makes HMMs particularly attractive for our task is that they allow us to combine two sources of information: first, the acoustic information present in the observed signal, and second, knowledge about likely and unlikely word combinations in a well-formed text. Both sources of information are important for recovering the original text.

To utilize HMMs for our task, we need to solve two problems: we need to estimate the model parameters of the HMM (training phase), and we need to determine the most likely sequence of hidden states for a sequence of observations given the model (recognition phase). The method described in Section 3.2 approximates the estimation of the emission probabilities by computing a ranking of the candidate words given an observed acoustic signal. The initial probabilities, which model the probability of starting in a given state, and the transition probabilities, which model the likelihood of different words following each other in an English text, can be obtained by building a *language model* from a large text corpus. To address the second problem, determining the most likely sequence of hidden states (i.e., the most likely sequence of printed words in the target text), we can use the Viterbi algorithm [37]. In the following two sections, we describe in more detail how we compute the language models and how the candidate words are reordered by applying the Viterbi algorithm.

### 3.3.2 Building the language models

A language model of size $n$ assigns a probability to each sequence of $n$ words. The probability distribution can be estimated by computing the frequencies of all $n$-grams from a large text corpus. Note that language models are to some extent domain and genre dependent, i.e., a language model built from a corpus of financial texts will not be a very good model for predicting likely word sequences in biomedical texts. To cover a large range of domains and thus make our model robust in the face of arbitrary input texts, we train the language model on a diverse selection of stable Wikipedia articles. The corpus has a size of 63 MB and contains approximately 10 million words. For our domain-specific experiments, we used a corpus of living-will declarations consisting of 14,000 words of English text. From the corpus, we extracted all 3-grams and computed their frequencies.[1] We took into consideration all 3-grams that appeared at least 3 times. As $n$-grams with probability 0 will never be selected by the Viterbi algorithm, we smooth the probabilities by assigning a small probability to each unseen $n$-gram.

The length of an $n$-gram determines how many words of context (i.e., how many previous hidden states in the HMM) are taken into account by the language model. Higher values for $n$ can lead to better models but also require exponentially larger corpora for an accurate estimation of the $n$-gram probabilities. The higher the value of $n$, the larger the likelihood that some $n$-grams never appear in the corpus, even though they are valid word sequences and thus may still appear in the printed text.

### 3.3.3 Reordering of candidate words based on language models

Having built the language model, we can reorder the candidate words using the model to select the most likely word sequence (i.e., the most likely sequence of hidden states). This task is addressed by the Viterbi algorithm [37], which takes as input an HMM $\langle Q, O, A, B, I \rangle$ of order $d$ and a sequence of observations $a_1, \ldots, a_T \in O^T$. Its state consists of $\Psi = T \times Q^d$. First, the $d$-th step is initialized (the earlier are unused) according to the initial distribution, weighted with the

[1] All 3-grams were converted to lower case and punctuation characters were stripped off.

observations:

$$\Psi_{d,i_1,\ldots,i_d} = I_{i_1,\ldots,i_d} \prod_{k=1,\ldots,d} B_{i_k,a_k} \quad \forall\, 1 \le i,j \le N.$$

In the recursion, for increasing indices $s$, the maximum of all previous values is taken:

$$\Psi_{s,i_1,\ldots,i_d} = B_{i_d,a_s} \max_{i_0 \in Q} \left( A_{i_0,i_1,\ldots,i_d} \Psi_{s-1,i_0,\ldots,i_{d-1}} \right) \quad \forall\, s > d, 1 \le i,j \le N.$$

Finally, the sequence of hidden states can be obtained by backtracking the indices that contributed to the maximum in the recursion step.

The memory required to store the state $\Psi$ is $O(T \cdot N^d)$, and the running time is $O(T \cdot N^{d+1})$, as we are optimizing over all $N$ hidden states for each cell, so memory requirements are a major challenge in implementing the Viterbi algorithm. For example, using a dictionary of $1,000$ words, the memory requirements of our implementation for 3-grams are slightly above 2 GB, and is growing quadratically in $N$.

We use two techniques to overcome these problems:

1. First, instead of storing the complete transition matrix $A$ we compute the values on-the-fly (keeping only the list of 3-grams in memory).

2. Second, we do not optimize over all possible words, but only over the $M = 30$ best rated words from the previous stage. This brings down memory requirements to $O(T \cdot M^d)$ and execution time to $O(T \cdot M^{d+1})$. The size of $\Psi$ in this case is 130 MB for 3-grams.

Further improvements are conceivable, e.g., by using parallel scalability [40].

## 4 Experiments and Statistical Evaluation

In this section we describe our experiments for evaluating the attack. In addition to describing the set-up and the experimental results on the recognition rate for sample articles, we present our experiments for evaluating the influence of using different microphones, printers, fonts, etc. on the recognition rate; moreover, we identify and evaluate countermeasures.

### 4.1 Setup

We use an Epson LQ-300+II (24 needles) without printer cover and the in-built mono-spaced font for printing texts. The sound is recorded from a short distance using a Sennheiser MKH-8040 microphone with nominal frequency range from 30 Hz to 50 kHz. If nothing additional is mentioned the experiments were conducted in a normal office with the door closed and no people talking inside the room. There was no special shielding against noise from the outside (e.g., traffic noise). In the training phase we used a dictionary containing 1,400 words; the dictionary consists of a list of the 1,000 most frequent words from our corpus augmented with the words that appeared in our example texts.[2] Inflected forms, capitalization, as well as words with leading punctuation marks need to be counted as different words, as their sound features might significantly differ (blurring propagates from left to right within a word).

We work with the sound recordings of four different articles from Wikipedia on different topics: two articles on computer science (on source-code and printers), one article on politics (on Barack Obama), and one article on art (on architecture) with a total of 1,181 words to evaluate the attack.

The training and matching phase have been implemented in MATLAB using the Signal Processing Toolbox – a MATLAB extension which allows to conveniently process audio signals. The HMM-based post-processing is implemented in C. The tool is fully automated, with the only exceptions being threshold values that need manual adaption for a given attack scenario. In the scenario with the microphone placed 10cm in front of the printer obtaining the threshold values is straightforward, as they can be determined directly from the intensity plots. In case of a more blurred signal (e.g., due to a larger distance), we iteratively determined suitable values, essentially by trial-and-error. The training phase takes a one-time effort of several hours for building up the sound feature database for the words in the dictionary. The recognition phase takes approximately 2 hours for matching one page of text, including full HMM-based post-processing. Memory usage of the procedure is substantial, because the feature database and the HMM-related information are kept in main memory to speed up computation. Trade-offs with less memory consumption but larger execution times can easily be realized.

### 4.2 Results

The recognition rates for the four articles in our experiments are depicted in Figure 6. The first row shows the recognition rates if no HMM-based post-processing is used, i.e., these numbers correspond to the output of the matching phase. For illustration, we wrote in brackets the rate that the correct word was within the three highest-ranked words in the matching phase. The second row depicts the recognition rates after applying post-processing with HMMs based on 3-grams. We thus achieve recognition rates between 67 % and 72 % for the four articles.

[2] In a real attack, ensuring that (almost) all words of the text occur in the dictionary can be achieved using several techniques: Using contextual knowledge to reduce the number of words that are likely to appear in the text, training a larger dictionary, or using feedback-based learning to subsequently add missing words to the dictionary.

| | Text 1 | Text 2 | Text 3 | Text 4 | **Overall** |
|---|---|---|---|---|---|
| Basic Top 1 (*Top 3*) | 60.5 % (*75.1 %*) | 66.5 % (*79.2 %*) | 62.8 % (*78.7 %*) | 61.5 % (*77.9 %*) | **62.9** % (*78.0 %*) |
| HMM 3-gram | 66.7 % | 71.8 % | 71.2 % | 69.0 % | **69.9** % |

Figure 6: Recognition rates of our four sample articles. The first row shows the recognition rates if no HMM-based post-processing is used; the second row depicts the recognition rates after applying post-processing with HMMs based on 3-grams using a general-purpose corpus.

| | Declaration 1 | Declaration 2 |
|---|---|---|
| Basic Top 1 (*Top 3*) | 59.5 % (*77.8 %*) | 57.5 % (*72.6 %*) |
| HMM 3-gram (using general-purpose corpus) | 68.3 % | 60.8 % |
| **HMM 3-gram (using domain-specific corpus)** | **95.2** % | **72.5** % |

Figure 7: Recognition rates of our two additional documents using domain-specific HMM-based post-processing. The first row shows the recognition rates without HMM-based post-processing; the second and third rows depict the recognition rates after applying post-processing with HMMs based on 3-grams using a general-purpose corpus and a domain-specific corpus, respectively.

While the aforementioned results employ HMM-based post-processing using a general-purpose corpus, our experiments indicate that domain-specific corpora yield even better results. Recall that we considered two additional documents containing living-will declarations that we intended to analyze using a domain-specific corpus. The recognition rates for the two living-will declarations are depicted in Figure 7. The first / second row again depict the results without / with general-purpose HMM-based post-processing; the third row shows the results for HMM-based post-processing using the domain-specific corpus. We achieve recognition rates of $95.2$ % and $72.5$ % for the two documents, respectively. Text examples for the reconstruction using a general-purpose corpus and a domain-specific corpus are provided in Appendix A and Appendix B, respectively.

We also experimented with 4-gram and 5-gram language models. In addition to encountering even more severe problems of memory consumptions, our experiments indicated that the recognition rates do not improve over 3-grams. While this behavior might be surprising at a first glance, it can be explained by the sparseness of the training data: The number of 5-grams that we can extract from our corpus is approx. $10^7$, but the transition matrix of an HMM based on 5-grams on a dictionary of 1,000 words has $10^{15}$ entries; thus the number of 5-grams is too small compared to the number of entries. For similar reasons 4-grams and 5-grams are rarely used in natural language processing.

### 4.3 Discussion and Supplemental Experiments

We have evaluated the influence on the recognition rate of using different microphones, different printers, proportional fonts, etc., and we investigated why the reconstruction works from a conceptual perspective. In a nutshell, the results can be summarized as follows (details are given below): Several parameters of modified set-ups did not affect the recognition rate and gave comparable results, e.g., using cheaper microphones or using different printers (of the same model) for the training phase and the recognition phase. Using proportional instead of mono-spaced fonts or using different printer models only slightly decreased the recognition rate. Some considerably stronger modifications, however, did not work out at all, and they can be seen as conceptual limitations of our attack. This comprises using completely different printer technologies such as ink-jet or laser printers (because of the absence of suitable sound emissions that can be used to mount the attack). We provide statistical results on these modifications below. Furthermore, we evaluate countermeasures.

#### 4.3.1 Using different microphones

Our experiments have indicated that information that is relevant for us is carried in the frequency range above approximately 20 kHz, see Section 3. Microphones with nominal frequency range higher than 20 kHz are rather expensive, e.g., the Sennheiser microphone referred to in Section 4.1 has a frequency range up to 50 kHz and costs approximately 1,300 dollars. However, our experiments have shown that some microphones with a nominal frequency range of 20 kHz are sensitive to higher fre-

| | Top 1 | (*Top 3*) |
|---|---|---|
| Sennheiser MKH-8040 microphone and Epson LQ-300+II printer | 62 % | (*78 %*) |
| Behringer B-5 microphone | 59 % | (*85 %*) |
| Sennheiser ME 2 clip-on microphone | 57 % | (*72 %*) |
| OKI Microline 1190 printer | 41 % | (*51 %*) |
| Another Epson LQ-300+II | 54 % | (*72 %*) |
| Proportional font | 57 % | (*71 %*) |

Figure 8: Results of the reconstruction with different microphone models and different printer models. (These control experiments were conducted on shorter texts and corpora than the previous experiments and no HMM-based post-processing was applied.)

quencies as well (possibly with less accurate frequency response, but this had no noticeable influence on the recognition rate as long as we use the same microphone for recording both the training data and the attack data). Figure 8 shows in the second row the recognition rates of one sample article if a Behringer B-5 microphone is used, which has a nominal frequency range up to 20 kHz and costs approximately 80 dollars. The results obtained with the Behringer microphone are only slightly worse than the results using the Sennheiser microphone.

We also conducted an experiment using a small clip-on microphone – a Sennheiser ME 2 with nominal frequency range up to 18 kHz, which costs approximately 130 dollars. The recognition rates of one sample article are shown in the third row of Figure 8; they are again only slightly worse than the rates with the larger Sennheiser microphone.

#### 4.3.2 Using different dot-matrix printers

We also evaluated if the printer model influences the recognition rate. The fourth row of Figure 8 shows the recognition rates of one article printed with an OKI Microline 1190 printer. The recognition rate is not as good as for the Epson printer, but it is still good.

So far we always considered the set-up that training data and the attacked text are printed on the same printer. In a realistic attack scenario, however, it is unlikely that the attacker can print the training data on the same printer, but instead arranges access to another printer of the same printer model that he places in an acoustically similar environment. Our in-field attack described in detail in Section 5 is of this kind.

We demonstrate that the recognition rate only decreases slightly when using a different printer in the training phase. For this experiment we used the feature database that we previously recorded in the experiment described in Section 4.2, and printed one article on another Epson LQ-300+II printer that we bought from a different vendor. The recognition rate is shown in Figure 8, indicating a decrease of recognition rate of about 8 % compared with the results from Section 4.2.

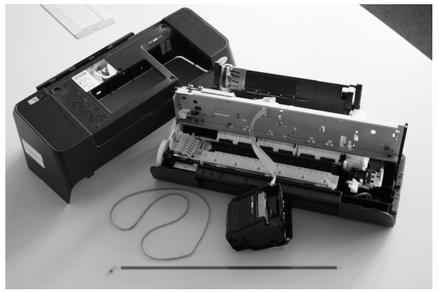


Figure 9: Ink-jet printer, disassembled for analysis.

This shows that it is practical to train a large dictionary offline. In the in-field attack described in Section 5 we use this result and train a dictionary on a separate printer.

#### 4.3.3 Using proportional fonts

Monospaced fonts are commonly used in many applications of dot-matrix printers; in particular, the in-built fonts are monospaced, and most applications seem to use these in-built fonts. Using proportional fonts instead intuitively relies on a more compact depiction of words that amplifies the effect of blurring. However, our experiments demonstrate that the recognition still works well, at a slightly lower rate (see Figure 8).

#### 4.3.4 On attacking other printer technologies

While dot-matrix printers are still deployed in some security-critical applications (see Figure 1), they have been replaced by other printer technologies such as ink-jet printers (see Figure 9) and laser printers in other applications. Ink-jet printers might be susceptible to similar attacks, as they construct the printout from individual dots, as dot-matrix printers do. On the one hand, the bubbles of ink might produce shock-waves in the air that potentially can be captured by a microphone; on the other hand, the piezo-electric elements used in some ink-jet printers might produce noise that can be measured. However, we were not able to capture these emanations. One reason might be that these faint sounds, if they exist, are dominated by the noise emitted by the mechanical parts of a printer. For laser printers, one expects that no information about the printed text is leaked, and our experiments support this view. Thus, to the best of our knowledge, these printer technologies seem to be unaffected by this kind of attack.

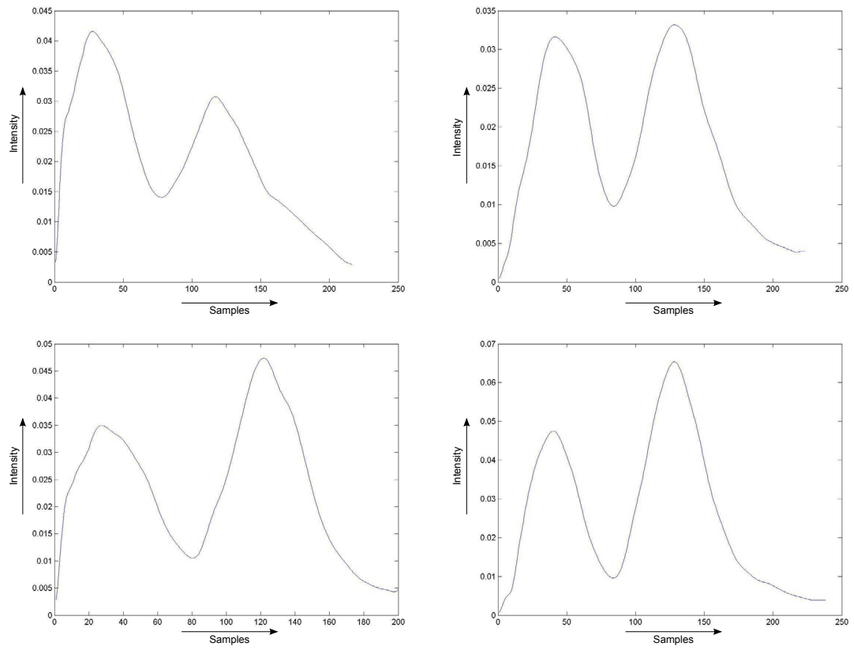



Figure 10: Each graph shows the intensity measured when printing a single vertical line, demonstrating the variations that can occur.

| | Top 1 | (*Top 3*) |
|---|---|---|
| Short distance, no cover | 62 % | (*78 %*) |
| With cover | 24 % | (*35 %*) |
| With foam box | 51 % | (*63 %*) |
| From 2 meters | 4 % | (*6 %*) |
| Closed door | 0 % | (*0 %*) |

Figure 11: More results of the reconstruction evaluating the effectiveness of different countermeasures. (These control experiments were conducted on a shorter text than the previous experiments, no HMM-based post-processing was applied.)

### 4.4 Countermeasures

The (obvious) idea that underlies all countermeasures is to suppress the acoustic emanations so far that reconstruction becomes hard in practical scenarios.

*Acoustic shielding foam:* The specific printer model that we used in most experiments has an optional printer cover with embedded acoustic shielding foam. Closing this cover absorbs a substantial amount of the acoustic emanation (see Figure 11). To further evaluate this idea, we built a box out of ordinary acoustic foam and placed the printer inside (shown in Figure 12). In contrast to the results with the cover, the recognition rate for the foam box was surprisingly good; 51 % of the words were reconstructed successfully. We believe that the shielding characteristics of the two types of foam suppress different ranges of the acoustic spectrum and thus have different effects on the reconstruction rate.

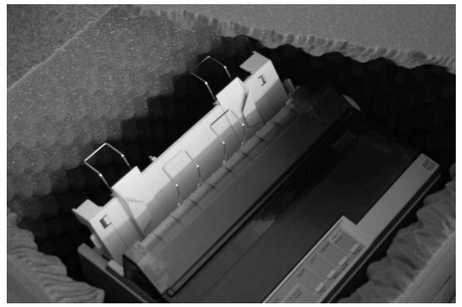


Figure 12: Printer in foam box for shielding evaluation.

Figure 13: The setup of the in-field attack.

*Distance:* Our experiments indicate that the recognition rate drops substantially if the distance between the printer and the microphone is increased. From a distance of 2 meters, the recognition rate drops to approximately 4 % (see Figure 11). From this distance our algorithm for splitting the signal into words requires manual intervention, as the audio signal contains more noise. However, we stress that this limitation can be circumvented in an in-field attack by placing a miniaturized wireless bug in close proximity to (or even in) the printer.

*Closed door:* We also tested the reconstruction from outside the printer's room with the door closed; the overall distance between the printer and the microphone was 4 meters. As expected, we found that in this setup no reconstruction was possible at all.

Our results indicate that ensuring the absence of microphones in the printer's room is sufficient to protect privacy. Unfortunately, this evaluation is not guaranteed to be complete; we merely state that our attack does not work under these circumstances. However, we believe that the potential for improvement is limited; thus the above discussion still provides reasonable estimates. As future work, we furthermore plan to investigate additional countermeasures such as introducing randomness into the printer's sound through software changes, e.g., by letting the printer print individual letters in a (somewhat) randomized order instead of always proceeding left-to-right.

## 5 In-field Attack

We have successfully mounted the attack in-field in a doctor's practice to recover the content of medical prescriptions (the setup of the attack is shown in Figure 13). For privacy reasons, we asked for permission upfront and let the secretary print fresh prescriptions of an artificial client. The attack was conducted under realistic – and arguably even pessimistic – circumstances: during rush hour, with many people chatting in the waiting room.

We recorded the emitted sounds of printing seven different prescriptions. We handed over all sound recordings, the printouts of six prescriptions, and a printer of the same type (an Epson LQ-570) that we bought at Ebay to one of the authors of this paper. The printouts were only used to determine which parts of the sound recording correspond to which parts of the prescription. The attack was carried out blindly, i.e, this author obtained no information about the seventh prescription except for its recorded sound.

The author carrying out the attack took the following steps:

1. From the available printouts, he first identified the position of the prescribed medication, the direction of printing, and the used font.
2. Using a suitable threshold, he subsequently determined the correct length and the white-space positions.
3. From a publicly available medication directory with about 14,000 different medications, he then determined possible candidates that matched these lengths. Here, abbreviations of words were also taken into account. The list of remaining candidates consisted of 29 entries.
4. The selection of candidate words (without HMM-based post-processing) then already revealed the correct medication out of the remaining 29 candidates.

   The correct medication was "Müller'sche Tabletten bei Halsschmerzen", a medication against sore throat. The printing was even abbreviated on the prescription as

   ```
   Müller'sche Tabletten bei
   Halsschm.
   ```



The attack was actually easier to conduct in this practical scenario compared to the experiments in Section 4, because we were able to substantially narrow down the list of candidates by taking into account length information of the medication. Admittedly, the secretary herself unintentionally simplified this task by selecting a long medication name consisting of several words.

## 6 Conclusion

We have presented a novel attack that takes as input a sound recording of a dot-matrix printer processing English text, and recovers up to 72 % of printed words. If we assume contextual knowledge about the text, the attack achieves recognition rates up to 95 %. After an upfront training phase, the attack is fully automated and uses a combination of machine learning, audio processing and speech recognition techniques, including spectrum features, Hidden Markov Models and linear classification; moreover, it allows for feedback-based incremental learning. We have identified and evaluated countermeasures that are suitable to prevent this kind of attack. We have successfully mounted the attack in-field in a doctor's practice to recover the content of medical prescriptions under realistic conditions. Moreover, we have shown the relevance of this attack by commissioning a representative survey that showed that dot-matrix printers are still deployed in a variety of sensitive areas, in particular by banks and doctors.

## References

[1] National Security Agency. TEMPEST: A signal problem. Available online at `http://www.nsa.gov/public/pdf/tempest.pdf`, 1972.

[2] Ross J. Anderson and Markus G. Kuhn. Soft tempest – an opportunity for NATO. In *Information Systems Technology (IST) Symposium "Protecting NATO Information Systems in the 21st Century"*, 1999.

[3] Dmitri Asonov and Rakesh Agrawal. Keyboard acoustic emanations. In *Proc. 2004 IEEE Symposium on Security and Privacy (Oakland 2004)*, pages 3–11, 2004.

[4] Michael Backes, Tongbo Chen, Markus Dürmuth, Hendrik P. A. Lensch, and Martin Welk. Tempest in a teapot: Compromising reflections revisited. In *Proc. 2009 IEEE Symposium on Security and Privacy (Oakland 2009)*, pages 315–327. IEEE Computer Society, 2009.

[5] Michael Backes, Markus Dürmuth, and Dominique Unruh. Compromising reflections – or – how to read LCD monitors around the corner. In *Proc. 2008 IEEE Symposium on Security and Privacy (Oakland 2008)*, pages 158–169. IEEE Computer Society, 2008.

[6] Arthur O. Bauer. Some aspects of military line communications as deployed by the german armed forces prior to 1945. In *The History of Military Communications, Proc. 5th Annual Colloquium*. Centre for the History of Defence Electronics, Bournemouth University, 1999.

[7] Yigael Berger, Avishai Wool, and Arie Yeredor. Dictionary attacks using keyboard acoustic emanations. In *Proc. 13th ACM Conference on Computer and Communication Security (CCS 2006)*, pages 245–254. ACM, 2006.

[8] Eckhard Beubler. *Schmerzbehandlung in der Palliativmedizin*, chapter Rezeptur in verschiedenen europäischen Ländern: gesetzliche Grundlagen, pages 249–255. Springer, 2006.

[9] Thomas L. Blum, Douglas F. Keislar, James A. Wheaton, and Erling H. Wold. United states patent 5918223: Method and article of manufacture for content-based analysis, storage, retrieval, and segmentation of audio information, 1999.

[10] Roland Briol. Emanation: How to keep your data confidential. In *Proc. Symposium on Electromagnetic Security for Information Protection*, 1991.

[11] Donald G. Childers, David P. Skinner, and Robert C. Kemerait. The cepstrum: A guide to processing. In *Proc. of the IEEE*, volume 65, pages 1428–1443, 1977.

[12] Kenneth W. Church. A stochastic parts program and noun phrase parser for unrestricted text. In *Proc. of ANLP-2*, pages 136–143, 1988.

[13] Trial Data Inc. Do your own will | free living will forms. Online at `http://www.doyourownwill.com/lw/aklw.doc`, 2010.

[14] Steven DeRose. Grammatical category disambiguation by statistical optimization. *Computational Linguistics*, 14(1):31–39, 1988.

[15] Jonathan T. Foote. Content-based retrieval of music and audio. In *Proc. of Multimedia Storage and Archiving*, volume 3229, pages 138–147, 1997.

[16] Harold Joseph Highland. Electromagnetic radiation revisited. *Comput. Secur.*, 5(2):85–93, 1986.

[17] Frederick Jelinek. *Statistical Models for Speech Recognition*. MIT Press, 1998.

[18] Biing-Hwang Juang and Lawrence R. Rabiner. Hidden markov models for speech recognition. *Technometrics*, 33(3):251–272, 1991.

[19] Markus G. Kuhn. Optical time-domain eavesdropping risks of CRT displays. In *Proc. 2002 IEEE Symposium on Security and Privacy (Oakland 2002)*, pages 3–18, 2002.

[20] Markus G. Kuhn. *Compromising Emanations: Eavesdropping Risks of Computer Displays*. PhD thesis, University of Cambridge, 2003.

[21] Markus G. Kuhn. Electromagnetic eavesdropping risks of flat-panel displays. In *Proc. 4th Workshop on Privacy Enhancing Technologies (PET 2005)*, pages 88–107, 2005.

[22] Markus G. Kuhn. Security limits for compromising emanations. In *Proc. 7th International Workshop of Cryptographic Hardware and Embedded Systems (CHES 2005)*, volume 3659 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2005.

[23] Beth Logan. Mel frequency cepstral coefficients for music modeling. In *Proc. 1st International Conference on Music Information Retrieval*, 2000.

[24] Beth Logan and Ariel Salomon. A music similarity function based on signal analysis. In *Proc. IEEE International Conference on Multimedia and Expo*. IEEE Computer Society, 2001.

[25] Joe Loughry and David A. Umphress. Information leakage from optical emanation. *ACM Transactions on Information and Systems Security*, 5(3):262–289, 2002.

[26] Isoplan :markforschung. Study on the usage of dot-matrix printers. Available Online at `http://dot-matrix-survey.webs.com/`, 2009.

[27] Meinard Müller. *Information Retrieval for Music and Motion*. Springer, 2007.

[28] Meinard Müller, Frank Kurth, and Michael Clausen. Audio matching via chroma-based statistical features. In *Proc. 6th International Conference on Music Information Retrieval*, pages 288–295, 2005.

[29] R. Nag, Kin Hong Wong, and Frank Fallside. Script recognition using Hidden Markov Models. In *Proc. International Conference on Acoustic Speech and Signal Processing*, pages 2071–2074. IEEE Computer Society, 1986.

[30] Francois Pachet and Jean-Julien Aucouturier. Music similarity measures: What's the use? In *Proc. 3rd International Conference on Music Information Retrieval*, 2002.

[31] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proc. of the IEEE*, 77(2):257–286, 1989.

[32] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. The earth mover's distance as a metric for image retrieval. *Int. Journal of Computer Vision*, 40(2):99–121, 2000.

[33] Adi Shamir and Eran Tromer. Acoustic cryptanalysis – On nosy people and noisy machines. Available online at `http://people.csail.mit.edu/tromer/acoustic/`.

[34] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[35] Peter Smulders. The threat of information theft by reception of electromagnetic radiation from RS-232 cables. *Computers & Security*, 9:53–58, 1990.

[36] Wim van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4:269–286, 1985.

[37] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–267, 1967.

[38] Wikipedia. Printer (computing) — wikipedia, the free encyclopedia. Online at `http://en.wikipedia.org/w/index.php?title=Printer_(computing)&oldid=247592038`, 2008.

[39] Peter Wright. *Spy Catcher: The Candid Autobiography of a Senior Intelligence Officer*. Viking Adult, 1987.

[40] Kisun You, Jike Chong, Youngmin Yi, Ekaterina Gonina, Christopher Hughes, Yen-Kuang Chen, Wonyong Sung, and Kurt Keutzer. Scalable HMM based inference engine in large vocabulary continuous speech recognition. *IEEE Signal Processing Magazine*, 2010.

[41] John Young. How old is tempest? Online response collection. Online at `http://cryptome.org/tempest-old.htm`, February 2000.



[42] Li Zhuang, Feng Zhou, and J. Doug Tygar. Keyboard acoustic emanations revisited. In *Proc. 12th ACM Conference on Computer and Communication Security (CCS 2005)*, pages 373–382. ACM Press, 2005.

# A Example Text Recognition with General-purpose HMM Post-processing

In the following we give an excerpt of the text on printers [38], see Section 4.2, to demonstrate the reconstruction.

## A.1 The original text

First, we give the original text.

```
In computing, a printer is a
peripheral which produces a hard
copy (permanent human-readable
text and/or graphics) of documents
stored in electronic form, usually
on physical print media such as
paper or transparencies.  Many
printers are primarily used
as local peripherals, and are
attached by a printer cable
or, in most newer printers, a
USB cable to a computer which
serves as a document source.
Some printers, commonly known
as network printers, have built-in
network interfaces (typically
wireless or Ethernet), and can
serve as a hardcopy device for any
user on the network.  Individual
printers are often designed to
support both local and network
connected users at the same time.
```

## A.2 Output of the reconstruction without HMM-based post-processing

Next, we give the reconstructed output without HMM-based post-processing. Recognition rate: 69 %.

```
In computing, a printer in 5
peripheral which produces 3 hard
body (permanent human-readable
text and/or graphics) of documents
status in electronic form.
usually 20 physical print media
Such 30 pages or transparencies.
Many Printers are primarily used
go local peripherals, end are
attached go A printer could
or, in most newer printers; =
USB cable go A computer which
served de = document source.
name printers, commonly known
go network printers; have built-in
network interfaces (typically
wireless As Ethernet), god way
serve As = hardcopy device for out
year we who network.  Individual
Printers use often designed 30
support born local god network
connected users go too name time.
```

## A.3 Output of the reconstruction with general-purpose HMM-based post-processing

Finally, we give the reconstructed output after applying the HMM-based post-processing using a general-purpose corpus. Recognition rate: 74 %.

```
in computing a printer in a
peripheral which produces a hard
body permanent human-readable
text and/or graphics of documents
source in electronic form usually
as physical print media such as
pages or transparencies many
printers are primarily used go
local peripherals end are attached
go a printer could or in most
newer printers a usb cable go
a computer which served de =
document source some printers
commonly known go network printers
have built-in network interfaces
typically wireless as ethernet god
way serve as a hardcopy device for
out year we who network individual
printers use often designed so
support born local god network
connected users as too some tree
```

# B Example Text Recognition with Domain-specific HMM Post-processing

In the following we illustrate the recognition of an excerpt of a living-will declaration [13], see Section 4.2, to illustrate the domain-specific post-processing.

## B.1 The original text

First, we give the original text.

```
ADVANCE HEALTH CARE DIRECTIVE
INSTRUCTIONS: This form lets you
give specific instructions about
any aspect of your health care.
Choices are provided for you to
express your wishes regarding
```

```
the provision, withholding,
or withdrawal of treatment to
keep you alive, as well as the
provision of pain relief.  Space
is provided for you to add to
the choices you have made or for
you to write out any additional
wishes.  This form also lets you
express an intention to donate
your bodily organs and tissues
following your death.  Lastly,
this form lets you designate
a physician to have primary
responsibility for your health
care.
```

```
express your wishes regarding
the provision withholding or
withdrawal of treatment to keep
you alive as well as the provision
of pain relief space is provided
for you to add to the choices
you have made or for you to david
out any additional wishes move
form also lets you express an
intention to donate your bodily
organs and tissues following your
death lastly this form lets you
designate a physician to have
primary responsibility for your
health care
```

### B.2 Output of the reconstruction with general-purpose HMM-based post-processing

Next, we give the reconstructed output of the general-purpose HMM-based post-processing. Recognition rate: 68 %.

```
advance health care directive
instructions only form into you
with consists observations peace
who appear on your health care
choices act provided for due to
century many witness according
one government declaration of
witnesses be competent to been
one alive as well as the provision
of pain primary power to provided
far one of out of now against
the once made of way and we allow
our own experience witness open
form with lets can average as
connected to donate year states
canada and tissues including heat
energy lastly this poor and you
designing b according to food
witness administration has been
health care
```

### B.3 Output of the reconstruction with domain-specific HMM-based post-processing

Finally, we give the reconstructed output after applying the HMM-based post-processing using a domain-specific corpus. Recognition rate: 95 %.

```
advance health care directive
instructions move form lets you
give consists instructions about
any aspect of your health care
choices are provided for you to
```



# Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study

*Ishtiaq Rouf[a], Rob Miller[b], Hossen Mustafa[a], Travis Taylor[a], Sangho Oh[b]*

*Wenyuan Xu[a], Marco Gruteser[b], Wade Trappe[b], Ivan Seskar[b] **

[a] *Dept. of CSE, Univ. of South Carolina, Columbia, SC USA*

{*rouf, mustafah, taylort9, wyxu*}*@cse.sc.edu*

[b] *WINLAB, Rutgers Univ., Piscataway, NJ USA*

{*rdmiller, sangho, gruteser, trappe, seskar*}*@winlab.rutgers.edu*

## Abstract

Wireless networks are being integrated into the modern automobile. The security and privacy implications of such in-car networks, however, have are not well understood as their transmissions propagate beyond the confines of a car's body. To understand the risks associated with these wireless systems, this paper presents a privacy and security evaluation of wireless Tire Pressure Monitoring Systems using both laboratory experiments with isolated tire pressure sensor modules and experiments with a complete vehicle system. We show that eavesdropping is easily possible at a distance of roughly 40m from a passing vehicle. Further, reverse-engineering of the underlying protocols revealed static 32 bit identifiers and that messages can be easily triggered remotely, which raises privacy concerns as vehicles can be tracked through these identifiers. Further, current protocols do not employ authentication and vehicle implementations do not perform basic input validation, thereby allowing for remote spoofing of sensor messages. We validated this experimentally by triggering tire pressure warning messages in a moving vehicle from a customized software radio attack platform located in a nearby vehicle. Finally, the paper concludes with a set of recommendations for improving the privacy and security of tire pressure monitoring systems and other forthcoming in-car wireless sensor networks.

## 1 Introduction

The quest for increased safety and efficiency of automotive transportation system is leading car makers to integrate wireless communication systems into automobiles. While vehicle-to-vehicle and vehicle-to-infrastructure systems [22] have received much attention, the first wireless network installed in every new vehicle is actually an in-vehicle sensor network: the tire pressure monitoring system (TPMS). The wide deployment of TPMSs in the United States is an outgrowth of the TREAD Act [35] resulting from the Ford-Firestone tire failure controversy [17]. Beyond preventing tire failure, alerting drivers about underinflated tires promises to increase overall road safety and fuel economy because proper tire inflation improves traction, braking distances, and tire rolling resistance. These benefits have recently led to similar legislation in the European Union [7] which mandates TPMSs on all new vehicles starting in 2012.

Tire Pressure Monitoring Systems continuously measure air pressure inside all tires of passenger cars, trucks, and multipurpose passenger vehicles, and alert drivers if any tire is significantly underinflated. While both direct and indirect measurement technologies exist, only direct measurement has the measurement sensitivity required by the TREAD Act and is thus the only one in production. A *direct measurement* system uses battery-powered pressure sensors inside each tire to measure tire pressure and can typically detect any loss greater than 1.45 psi [40]. Since a wired connection from a rotating tire to the vehicle's electronic control unit is difficult to implement, the sensor module communicates its data via a radio frequency (RF) transmitter. The receiving tire pressure control unit, in turn, analyzes the data and can send results or commands to the central car computer over the Controller-area Network (CAN) to trigger a warning message on the vehicle dashboard, for example. *Indirect measurement* systems infer pressure differences between tires from differences in the rotational speed, which can be measured using the anti-lock braking system (ABS) sensors. A lower-pressure tire has to rotate faster to travel the same distance as a higher-pressure tire. The disadvantages of this approach are that it is less accurate, requires calibration by the driver, and cannot detect the simultaneous loss of pressure from all tires (for example, due to temperature changes). While initial versions of the TREAD Act allowed indirect technology, updated rul-

*This study was supported in part by the US National Science Foundation under grant CNS-0845896, CNS-0845671, and Army Research Office grant W911NF-09-1-0089.

ings by the United States National Highway Transportation Safety Administration (NHTSA) have required all new cars sold or manufactured after 2008 in the United States to be equipped with direct TPMS [35] due to these disadvantages.

## 1.1 Security and Privacy Risks

Security and privacy aspects of vehicle-to-vehicle and vehicle-to-infrastructure communication have received significant consideration by both practitioners and researchers [3, 36]. However, the already deployed in-car sensor communication systems have received little attention, because (i) the short communication range and metal vehicle body may render eavesdropping and spoofing attacks difficult and (ii) tire pressure information appears to be relatively innocuous. While we agree that the safety-critical application scenarios for vehicle-to-vehicle communications face higher security and privacy risks, we believe that even current tire pressure measurement systems present potential for misuse.

First, wireless devices are known to present tracking risks through explicit identifiers in protocols [20] or identifiable patterns in waveforms [10]. Since automobiles have become an essential element of our social fabric — they allow us to commute to and from work; they help us take care of errands like shopping and taking our children to day care — tracking automobiles presents substantial risks to location privacy. There is significant interest in wireless tracking of cars, at least for traffic monitoring purposes. Several entities are using mobile toll tag readers [4] to monitor traffic flows. Tracking through the TPMS system, if possible, would raise greater concerns because the use of TPMS is not voluntary and they are hard to deactivate.

Second, wireless is easier to jam or spoof because no physical connection is necessary. While spoofing a low tire pressure readings does not appear to be critical at first, it will lead to a dashboard warning and will likely cause the driver to pull over and inspect the tire. This presents ample opportunities for mischief and criminal activities, if past experience is any indication. Drivers have been willing to tinker with traffic light timing to reduce their commute time [6]. It has also been reported that highway robbers make drivers pull over by puncturing the car tires [23] or by simply signaling a driver that a tire problem exists. If nothing else, repeated false alarms will undermine drivers' faith in the system and lead them to ignore subsequent TPMS-related warnings, thereby making the TMPS system ineffective.

To what extent these risks apply to TPMS and more generally to in-car sensor systems remains unknown. A key question to judge these risks is whether the range at which messages can be overheard or spoofed is large enough to make such attacks feasible from outside the vehicle. While similar range questions have recently been investigated for RFID devices [27], the radio propagation environment within an automobile is different enough to warrant study because the metal body of a car could shield RF from escaping or entering a car. It is also unclear whether the TPMS message rate is high enough to make tracking vehicles feasible. This paper aims to fill this void, and presents a security and privacy analysis of state-of-the art commercial tire pressure monitoring systems, as well as detailed measurements for the communication range for in-car sensor transmissions.

## 1.2 Contributions

Following our experimental analysis of two popular TPMSs used in a large fraction of vehicles in the United States, this paper presents the following contributions:

**Lack of security measures.** TPMS communications are based on standard modulation schemes and simple protocols. Since the protocols do not rely on cryptographic mechanisms, the communication can be reverse-engineered, as we did using GNU Radio [2] in conjunction with the Universal Software Radio Peripheral (USRP) [1], a low-cost public software radio platform. Moreover, the implementation of the in-car system appears to fully trust all received messages. We found no evidence of basic security practices, such as input validation, being followed. Therefore, spoofing attacks and battery drain attacks are made possible and can cause TPMS to malfunction.

**Significant communication range.** While the vehicle's metal body does shield the signal, we found a larger than expected eavesdropping range. TPMS messages can be correctly received up to 10m from the car with a cheap antenna and up to 40m with a basic low noise amplifier. This means an adversary can overhear or spoof transmissions from the roadside or possibly from a nearby vehicle, and thus the transmission powers being used are not low enough to justify the lack of other security measures.

**Vehicle tracking.** Each in-tire sensor module contains a 32-bit immutable identifier in every message. The length of the identifier field renders tire sensor module IDs sufficiently unique to track cars. Although tracking vehicles is possible through vision-based automatic license plate identification, or through toll tag or other wireless car components, tracking through TPMS identifiers raises new concerns, because these transmitters are difficult for drivers to deactivate as they are available in all new cars and because wireless tracking is a low-cost solution compared to employing vision technology.

**Defenses.** We discuss security mechanisms that are applicable to this low-power in-car sensor scenario without taking away the ease of operation when installing a new tire. The mechanisms include relatively straightforward design changes in addition to recommendations for cryptographic protocols that will significantly mitigate TMPS security risks.

The insights obtained can benefit the design of other emerging wireless in-car sensing systems. Modern automobiles contain roughly three miles of wire [31], and this will only increase as we make our motor vehicles more intelligent through more on-board electronic components, ranging from navigation systems to entertainment systems to in-car sensors. Increasing the amount of wires directly affects car weight and wire complexity, which decreases fuel economy [13] and imposes difficulties on fault diagnosis [31]. For this reason, wireless technologies will increasingly be used in and around the car to collect control/status data of the car's electronics [16, 33]. Thus, understanding and addressing the vulnerabilities associated with internal automotive communications, and TPMS in particular, is essential to ensuring that the new wave of intelligent automotive applications will be safely deployed within our cars.

## 1.3 Outline

We begin in Section 2 by presenting an overview of TPMS and raising related security and privacy concerns. Although the specifics of the TPMS communication protocols are proprietary, we present our reverse-engineering effort that reveals the details of the protocols in Section 3. Then, we discuss our study on the susceptibility of TPMS to eavesdropping in Section 4 and message spoofing attacks in Section 5. After completing our security and privacy analysis, we recommend defense mechanisms to secure TPMS in Section 6. Finally, we wrap up our paper by presenting related work in Section 7 before concluding in Section 8.

# 2 TPMS Overview and Goals

**TPMS architecture.** A typical direct TPMS contains the following components: TPM sensors fitted into the back of the valve stem of each tire, a TPM electric control unit (ECU), a receiving unit (either integrated with the ECU or stand-alone), a dashboard TPM warning light, and one or four antennas connected to the receiving unit. The TPM sensors periodically broadcast the pressure and temperature measurements together with their

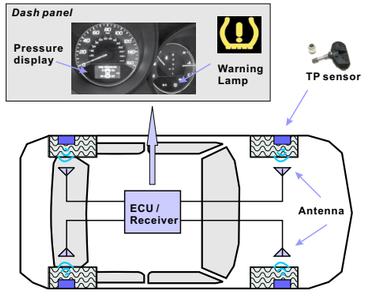



Figure 1: TPMS architecture with four antennas.

identifiers. The TPM ECU/receiver receives the packets and performs the following operations before sending messages to the TPM warning light. First, since it can receive packets from sensors belonging to neighboring cars, it filters out those packets. Second, it performs temperature compensation, where it normalizes the pressure readings and evaluates tire pressure changes. The exact design of the system differs across suppliers, particularly in terms of antenna configuration and communication protocols. A four-antenna configuration is normally used in high-end car models, whereby an antenna is mounted in each wheel housing behind the wheel arch shell and connected to a receiving unit through high frequency antenna cables, as depicted in Figure 1. The four-antenna system prolongs sensor battery life, since the antennas are mounted close to the TPM sensors which reduces the required sensor transmission power. However, to reduce automobile cost, the majority of car manufactories use one antenna, which is typically mounted on the rear window [11, 39].

**Communication protocols.** The communications protocols used between sensors and TPM ECUs are proprietary. From supplier websites and marketing materials, however, one learns that TPMS data transmissions commonly use the 315 MHz or 433 MHz bands (UHF) and ASK (Amplitude Shift Keying) or FSK (Frequency Shift Keying) modulation. Each tire pressure sensor carries an identifier (ID). Before the TPMS ECU can accept data reported by tire pressure sensors, IDs of the sensor and the position of the wheel that it is mounted on have to be entered to the TPMS ECU either manually in most cars or automatically in some high-end cars. This is typically done during tire installation. Afterwards, the ID of the sensor becomes the key information that assists the ECU in determining the origin of the data packet and filtering out packets transmitted by other vehicles.

To prolong battery life, tire pressure sensors are designed to sleep most of the time and wake up in two scenarios: (1) when the car starts to travel at high speeds (over 40 km/h), the sensors are required to monitor tire

pressures; (2) during diagnosis and the initial sensor ID binding phases, the sensors are required to transmit their IDs or other information to facilitate the procedures. Thus, the tire pressure sensors will wake up in response to two triggering mechanisms: a speed higher than 40 km/h detected by an on-board accelerometer or an RF activation signal.

The RF activation signals operate at 125 kHz in the low frequency (LF) radio frequency band and can only wake up sensors within a short range, due to the generally poor characteristics of RF antennas at that low frequency. According to manuals from different tire sensor manufacturers, the activation signal can be either a tone or a modulated signal. In either case, the LF receiver on the tire sensor filters the incoming activation signal and wakes up the sensor only when a matching signal is recognized. Activation signals are mainly used by car dealers to install and diagnose tire sensors, and are manufacturer-specific.

## 2.1 Security and Privacy Analysis Goals

Our analysis will concentrate on tracking risks through eavesdropping on sensor identifiers and on message spoofing risks to insert forged data in the vehicle ECU. The presence of an identifier raises the specter of location privacy concerns. If the sensor IDs were captured at roadside tracking points and stored in databases, third parties could infer or prove that the driver has visited potentially sensitive locations such as medical clinics, political meetings, or nightclubs. A similar example is seen with electronic toll records that are captured at highway entry and exit points by private entities for traffic monitoring purposes. In some states, these records are frequently subpoenaed for civil lawsuits. If tracking through the tire pressure monitoring system were possible, this would create additional concerns, particularly because the system will soon be present in all cars and cannot easily be deactivated by a driver.

Besides these privacy risks, we will consider attacks where an adversary interferes with the normal operations of TPMS by actively injecting forged messages. For instance, an adversary could attempt to send a low pressure packet to trigger a low pressure warning. Alternatively, the adversary could cycle through a few forged low pressure packets and a few normal pressure packets, causing the low pressure warning lights to turn on and off. Such attacks, if possible, could undermine drivers' faith in the system and potentially lead them to ignore TPMS-related warnings completely. Last but not least, since the TPM sensors always respond to the corresponding activation signal, an adversary that continuously transmits activation signals can force the tire sensors to send packets constantly, greatly reducing the lifetime of TPMS.

To evaluate the privacy and security risks of such a system, we will address the issues listed below in the following sections.

**Difficulty of reverse engineering.** Many potential attackers are unlikely to have access to insider information and must therefore reconstruct the protocols, both to be able to extract IDs to track vehicles and to spoof messages. The level of information necessary differs among attacks; replays for example might only require knowledge of the frequency band but more sophisticated spoofing requires protocol details. For spoofing attacks we also consider whether off-the-shelf radios can generate and transmit the packets appropriately.

**Identifier characteristics.** Tracking requires observing identifying characteristics from a message, so that multiple messages can be linked to the same vehicle. The success of tracking is closely tied to the answers to: (1) Are the sensor IDs used temporarily or over long time intervals? (2) Does the length of the sensor ID suffice to uniquely identify a car? Since the sensor IDs are meant to primarily identify their positions in the car, they may not be globally unique and may render tracking difficult.

**Transmission range and frequency.** Tracking further depends on whether a road-side tracking unit will be likely to overhear a transmission from a car passing at high speed. This requires understanding the range and messaging frequency of packet transmissions. To avoid interference between cars and to prolong the battery life, the transmission powers of the sensors are deliberately chosen to be low. Is it possible to track vehicles with such low transmission power combined with low messaging frequency?

**Security measures.** The ease of message spoofing depends on the use of security measures in TPMSs. The key questions to make message spoofing a practical threat include: (1) Are messages authenticated? (2) Does the vehicle use consistency checks and filtering mechanisms to reject suspicious packets? (3) How long, if possible, does it take the ECU to completely recover from a spoofing attack?

# 3 Reverse Engineering TPMS Communication Protocols

Analyzing security and privacy risks begins with obtaining a thorough comprehension of the protocols for *specific* sensor systems. To elaborate, one needs to know the modulation schemes, encoding schemes, and message formats, in addition to the activation and reporting



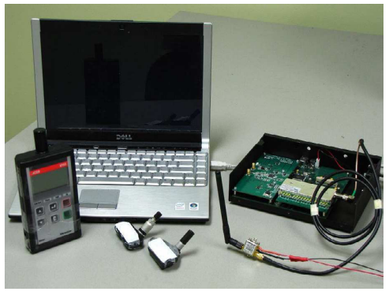


Figure 2: Equipment used for packet sniffing. At the bottom, from left to right are the ATEQ VT55 TPMS trigger tool, two tire pressure sensors (TPS-A and TPS-B), and a low noise amplifier (LNA). At the top is one laptop connected with a USRP with a TVRX daughterboard attached.

methodologies to properly decode or spoof sensor messages. Apart from access to an insider or the actual specifications, this information requires reverse-engineering by an adversary. To convey the level of difficulty of this process for in-car sensor protocols, we provide a brief walk-through of our approach below, where we begin by presenting relevant hardware.

**Tire pressure sensor equipment.** We selected two representative tire pressure sensors that employ different modulation schemes. Both sensors are used in automobiles with high market shares in the US. To prevent misuse of the information here, we refer to these sensors simply as *tire pressure sensor A (TPS-A)* and *tire pressure sensor B (TPS-B)*. To help our process, we also acquired a *TPMS trigger tool*, which is available for a few hundred dollars. Such tools are handheld devices that can activate and decode information from a variety of tire sensor implementations. These tools are commonly used by car technicians and mechanics for troubleshooting. For our experiments, we used a TPMS trigger tool from ATEQ [8] (ATEQ VT55).

**Raw signal sniffer.** Reverse engineering the TPMS protocols requires the capture and analysis of raw signal data. For this, we used GNU Radio [2] in conjunction with the Universal Software Radio Peripheral (USRP) [1]. GNU Radio is an open source, free software toolkit that provides a library of signal processing blocks that run on a host processing platform. Algorithms implemented using GNU Radio can receive data directly from the USRP, which is the hardware that provides RF access via an assortment of daughterboards. They include the TVRX daughterboard capable of receiving RF in the range of 50 Mhz to 870 MHz and the LFRX daughterboard able to receive from DC to 30 MHz. For convenience, we initially used an Agilent 89600 Vector Signal Analyzer (VSA) for data capture (but such equipment is not necessary). The pressure sensor modules, trigger tool, and software radio platform are shown in Figure 2.

## 3.1 Reverse Engineering Walk Through

While our public domain search resulted in only high-level knowledge about the TPM communication protocol specifics, anticipating sensor activity in the 315/433 MHz bands did provide us with a starting point for our reverse engineering analysis.

We began by collecting a few transmissions from each of the TPM sensors. The VSA was used to narrow down the spectral bandwidth necessary for fully capturing the transmissions. The sensors were placed close to the VSA receiving antenna while we used the ATEQ VT55 to trigger the sensors. Although initial data collections were obtained using the VSA, the research team switched to using the USRP to illustrate that our findings (and subsequently our attacks) can be achieved with low-cost hardware. An added benefit of using the USRP for the data collections is that it is capable of providing synchronized collects for the LF and HF frequency bands — thus allowing us to extract important timing information between the activation signals and the sensor responses. To perform these collects, the TVRX and LFRX daughterboards were used to provide access to the proper radio frequencies. Once the sensor bursts were collected, we began our signal analysis in MATLAB to understand the modulation and encoding schemes. The final step was to map out the message format.

**Determine coarse physical layer characteristics.** The first phase of characterizing the sensors involved measuring burst widths, bandwidth, and other physical layer properties. We observed that burst widths were on the order of 15 ms. During this initial analysis, we noted that each sensor transmitted multiple bursts in response to their respective activation signals. TPS-A used 4 bursts, while TPS-B responded with 5 bursts. Individual bursts in the series were determined to be exact copies of each other, thus each burst encapsulates a complete sensor report.

**Identify the modulation scheme.** Analysis of the baseband waveforms revealed two distinct modulation schemes. TPS-A employed amplitude shift keying (ASK), while TPS-B employed a hybrid modulation scheme — simultaneous usage of ASK and frequency shift keying (FSK). We speculate that the hybrid scheme is used for two reasons: (1) to maximize operability with TPM readers and (2) to mitigate the effects of an adverse channel during normal operation. Figure 3 illustrates the differences between the sensors' transmission in both the time and frequency domains. The modulation schemes are also observable in these plots.

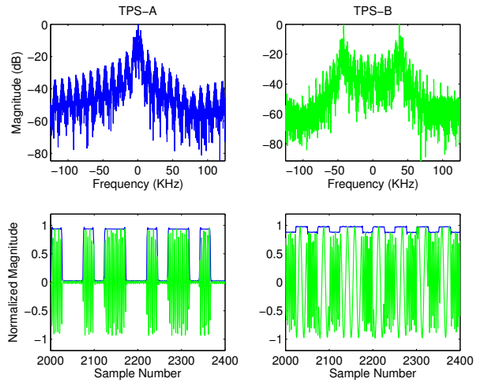



Figure 3: A comparison of FFT and signal strength time series between TSP-A and TSP-B sensors.

**Resolve the encoding scheme.** Despite the different modulation schemes, it was immediately apparent that both sensors were utilizing Manchester encoding (after distinct preamble sequences). The baud rate is directly observable under Manchester encoding and was on the order of 5 kBd. The next step was to determine the bit mappings from the Manchester encoded signal. In order to accomplish this goal, we leveraged knowledge of a known bit sequence in each message. We knew the sensor ID because it was printed on each sensor and assumed that this bit sequence must be contained in the message. We found that applying differential Manchester decoding generated a bit sequence containing the sensor ID.

**Reconstructing the message format.** While both sensors used differential Manchester encoding, their packet formats differed significantly. Thus, our next step was to determine the message mappings for the rest of the bits for each sensor. To understand the size and meaning of each bitfield, we manipulated sensor transmissions by varying a single parameter and observed which bits changed in the message. For instance, we adjusted the temperature using hot guns and refrigerators, or adjusted the pressure. By simultaneously using the ATEQ VT55, we were also able to observe the actual transmitted values and correlate them with our decoded bits. Using this approach, we managed to determine the majority of message fields and their meanings for both TPS-A and TPS-B. These included temperature, pressure, and sensor ID, as illustrated in Figure 4. We also identified the use of a CRC checksum and determined the CRC polynomials through a brute force search.

At this point, we did not yet understand the meaning of a few bits in the message. We were later able to reconstruct these by generating messages with our software radio, changing these bits, and observing the output of the


Figure 4: An illustration of a packet format. Note the size is not proportional to real packet fields.

TPMS tool or a real car. It turned out that these were parameters like battery status, over which we had no direct control by purely manipulating the sensor module. More details on message spoofing are presented in Section 5.

## 3.2 Lessons Learned

The aforementioned reverse-engineering can be accomplished with a reasonable background in communications and computer engineering. It took a few days for a PhD-level engineer experienced with reverse engineering to build an *initial* system. It took several weeks for an MS-level student with no prior experience in reverse engineering and GNU Radio programming to understand and reproduce the attack. The equipment used (the VTEQ VT55 and USRP attached with TVRX) is openly available and costs $1500 at current market prices.

Perhaps one of the most difficult issues involved baud rate estimation. Since Manchester encoding is used, our initial baud rate estimates involved averaging the gaps between the transition edges of the signal. However, the jitter (most likely associated with the local oscillators of the sensors) makes it almost impossible to estimate a baud rate accurate enough for a simple software-based decoder to work correctly. To address this problem, we modified our decoders to be self-adjustable to compensate for the estimation errors throughout the burst.

The reverse engineering revealed the following observations. First, it is evident that encryption has not been used—which makes the system vulnerable to various attacks. Second, each message contains a 28-bit or 32-bit sensor ID depending on the type of sensor. Regardless of the sensor type, the IDs do not change during the sensors' lifetimes.

Given that there are 254.4 million registered passenger vehicles in United States [34], one 28-bit Sensor ID is enough to track each registered car. Even in the future when the number of cars may exceed 256 million, we can still identify a car using a collection of tire IDs — a 4-tuple of tire IDs. Assuming a uniform distribution across the 28-bit ID space, the probability of an exact match of two cars' IDs is $4!/2^{112}$ without considering the ordering. To determine how many cars $R$ can be on the road in the US with a guarantee that there is a less than $P$ chance of any two or more cars having the same ID-set, is a classical birthday problem calculation:

$$R = \sqrt{\frac{2^{113}}{4!} ln(\frac{1}{1-P})}$$



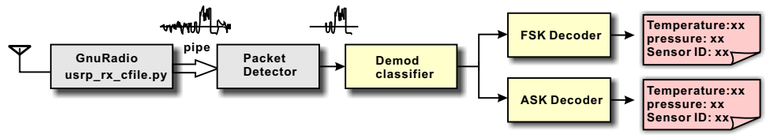



Figure 5: Block chart of the live decoder/eavesdropper.

To achieve a match rate of larger than $P = 1\%$, more than $10^{15}$ cars need to be on the road, which is significantly more than 1 billion cars. This calculation, of course, is predicated on the assumption of a uniform allocation across the 28-bit ID space. Even if we relax this assumption and assume 20 bits of entropy in a single 28-bit ID space, we would still need roughly 38 billion cars in the US to get a match rate of more than $P = 1\%$.

We note that this calculation is based on the unrealistic assumption that all 38 billion cars are co-located, and are using the same modulation and coding schemes. Ultimately, it is very unlikely to have two cars that would be falsely mistaken for each other.

# 4 Feasibility of Eavesdropping

A critical question for evaluating privacy implications of in-car wireless networks is whether the transmissions can be easily overheard from outside the vehicle body. While tire pressure data does not require strong confidentiality, the TPMS protocols contain identifiers that can be used to track the locations of a device. In practice, the probability that a transmission can be observed by a stationary receiver depends not only on the communication range but also on the messaging frequency and speed of the vehicle under observation, because these factors affect whether a transmission occurs in communication range.

The transmission power of pressure sensors is relatively small to prolong sensor battery lifetime and reduce cross-interference. Additionally, the NHTSA requires tire pressure sensors to transmit data only once every 60 seconds to 90 seconds. The low transmission power, low data report rate, and high travel speeds of automobiles raise questions about the feasibility of eavesdropping.

In this section, we experimentally evaluate the range of TPMS communications and further evaluate the feasibility of tracking. This range study will use TPS-A sensors, since their TPMS uses a four-antenna structure and operates at a lower transmission power. It should therefore be more difficult to overhear.

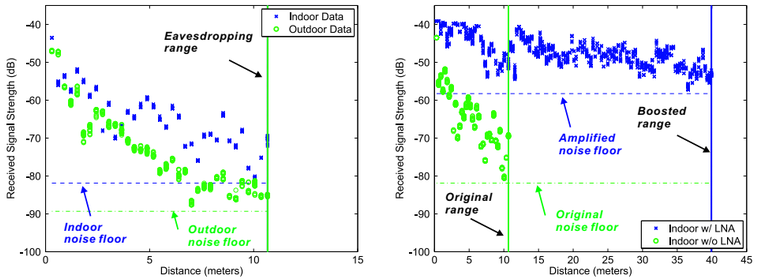
## 4.1 Eavesdropping System

During the reverse engineering steps, we developed two Matlab decoders: one for decoding ASK modulated TPS-A and the other for decoding the FSK modulated TPS-B. In order to reuse our decoders yet be able to constantly monitor the channel and only record useful data using GNU radio together with the USRP, we created a live decoder/eavesdropper leveraging pipes. We used the GNU Radio standard Python script `usrp_rx_cfile.py` to sample channels at a rate of 250 kHz, where the recorded data was then piped to a packet detector. Once the packet detector identifies high energy in the channel, it extracts the complete packet and passes the corresponding data to the decoder to extract the pressure, temperature, and the sensor ID. If decoding is successful, the sensor ID will be output to the screen and the raw packet signal along with the time stamp will be stored for later analysis. To be able to capture data from multiple different TPMS systems, the eavesdropping system would also need a modulation classifier to recognizes the modulation scheme and choose the corresponding decoder. For example, Liedtke's [29] algorithm could be used to differentiate ASK2 and FSK2. Such an eavesdropping system is depicted in Fig. 5.

In early experiments, we observed that the decoding script generates much erratic data from interference and artifacts of the dynamic channel environment. To address this problem, we made the script more robust and added a filter to discard erroneous data. This filter drops all signals that do not match TPS-A or TPS-B. We have tested our live decoder on the interstate highway I-26 (Columbia, South Carolina) with two cars running in parallel at speeds exceeding 110 km/h.

## 4.2 Eavesdropping Range

We measured the eavesdropping range in both indoor and outdoor scenarios by having the ATEQ VT55 trigger the sensors. In both scenarios, we fixed the location of the USRP at the origin $(0, 0)$ in Figure 7 and moved the sensor along the y-axis. In the indoor environment, we studied the reception range of stand-alone sensors in a hallway. In the outdoor environment, we drove one of the authors' cars around to measure the reception range of the sensors mounted in its front left wheel while the car's body was parallel to the x-axis, as shown in Figure 7. In our experiment, we noticed that we were able to decode the packets when the received signal strength is larger than the ambient noise floor. The resulting signal strength over the area where packets could be decoded




(a) indoor vs. outdoor (w/o LNA) (b) with LNA vs. without LNA (indoor)

Figure 6: Comparison of eavesdropping range of TPS-A.

successfully and the ambient noise floors are depicted in Figure 6 (a). The results show that both the outdoor and indoor eavesdropping ranges are roughly 10.7 m, the vehicle body appears only to have a minor attenuation effect with regard to a receiver positioned broadside.

We next performed the same set of range experiments while installing a low noise amplifier (LNA) between the antenna and the USRP radio front end, as shown in Figure 2. As indicated in Figure 6, the signal strength of the sensor transmissions still decreased with distance and the noise floor was raised because of the LNA, but the LNA amplified the received signal strength and improved the decoding range from 10.7 meters to 40 meters. This shows that with some inexpensive hardware a significant eavesdropping range can be achieved, a range that allows signals to be easily observed from the roadside.

Note that other ways to boost receiving range exist. Examples include the use of directional antennas or more sensitive omnidirectional antennas. We refer readers to the antenna studies in [9, 15, 42] for further information.

## 4.3 Eavesdropping Angle Study

We now investigate whether the car body has a larger attenuation effect if the receiver is located at different angular positions. We also study whether one USRP is enough to sniff packets from all four tire sensors.

**The effect of car body.** In our first set of experiments, we studied the effect of the car's metallic body on signal attenuation to determine the number of required USRPs. We placed the USRP antenna at the origin of the coordinate, as shown in Figure 7, and position the car at several points on the line of $y = 0.5$ with its body parallel to the x-axis. Eavesdropping at these points revealed that it is very hard to receive packets from four tires simultaneously. A set of received signal strength (RSS) measurements when the front left wheel was located at $(0, 0.5)$ meters are summarized in Table 1. Results show that the USRP can receive packets transmitted by the front left, front right and rear left sensors, but not from the rear right sensor due to the signal degradation caused by the car's metallic body. Thus, to assure receiving packets from all four sensors, at least two observation spots may be required, with each located on either side of the car. For instance, two USRPs can be placed at different spots, or two antennas connected to the same USRP can be meters apart.

**The eavesdropping angle at various distances.** We studied the range associated with one USRP receiving packets transmitted by the front left wheel. Again, we placed the USRP antenna at the origin and recorded packets when the car moved along trajectories parallel to the x-axis, as shown in Figure 7. These trajectories were 1.5 meters apart. Along each trajectory, we recorded RSS at the locations from where the USRP could decode packets. The colored region in Figure 11, therefore, denotes the eavesdropping range, and the contours illustrate the RSS distribution of the received packets.

From Figure 11, we observe that the maximum horizontal eavesdropping range, $r_{max}$, changes as a function of the distance between the trajectory and the USRP antenna, $d$. Additionally, the eavesdropping ranges on both sides of the USRP antenna are asymmetric due to the car's metallic body. Without the reflection and impediment of the car body, the USRP is able to receive the packets at further distances when the car is approaching rather than leaving. The numerical results of $r_{max}$, $\varphi_1$, the maximum eavesdropping angle when the car is approaching the USRP, and $\varphi_2$, the maximum angle when the car is leaving the USRP, are listed in Figure 8. Since

| Location | RSS (dB) | Location | RSS (dB) |
|---|---|---|---|
| Front left | -41.8 | Rear left | -55.0 |
| Front right | -54.4 | Rear right | N/A |

Table 1: RSS when USPR is located 0.5 meters away from the front left wheel.



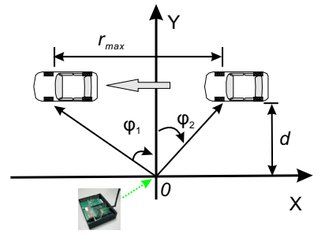



Figure 7: The experiment setup for the range study.

the widest range of 9.1 meters at the parallel trajectory was 3 meters away from the x-axis, an USRP should be placed 2.5 meters away from the lane marks to maximize the chance of packet reception, assuming cars travel 0.5 meter away from lane marks.

**Messaging rate.** According to NHTSA regulations, TPMS sensors transmit pressure information every 60 to 90 seconds. Our measurements confirmed that both TPS-A and TPS-B sensors transmit one packet every 60 seconds or so. Interestingly, contrary to documentation (where sensors should report data periodically after a speed higher than 40 km/h), both sensors periodically transmit packet even when cars are stationary. Furthermore, TPS-B transmits periodic packets even when the car is not running.

## 4.4 Lessons Learned: Feasibility of Tracking Automobiles

The surprising range of 40m makes it possible to capture a packet and its identifiers from the roadside, if the car is stationary (e.g., a traffic light or a parking lot). Given that a TPMS sensor only send one message per minute, tracking becomes difficult at higher speeds. Consider, for example, a passive tracking system deployed along the roadside at highway entry and exit ramps, which seeks to extract the unique sensor ID for each car and link entry and exit locations as well as subsequent trips. To ensure capturing at least one packet, a row of sniffers would be required to cover the stretch of road that takes a car 60 seconds to travel. The number of required sniffers, $n_{passive} = ceil(v * T/r_{max})$, where $v$ is the speed of the vehicle, T is the message report period, and $r_{max}$ is the detection range of the sniffer. Using the sniffing system described in previous sections where $r_{max} = 9.1$ m, 110 sniffers are required to guarantee capturing one packet transmitted by a car traveling at 60 km/h. Deploying such a tracking system appears cost-prohibitive.

It is possible to track with fewer sniffers, however, by leveraging the activation signal. The tracking station can send the 125kHz activation signal to trigger a transmission by the sensor. To achieve this, the triggers and sniffers should be deployed in a way such that they meet the following requirements regardless of the cars' travel speeds: (1) the transmission range of the trigger should be large enough so that the passing car is able to receive the complete activation signal; (2) the sniffer should be placed at a distance from the activation sender so that the car is in the sniffers' eavesdropping range when it starts to transmit; and (3) the car should stay within the eavesdropping range before it finishes the transmission.

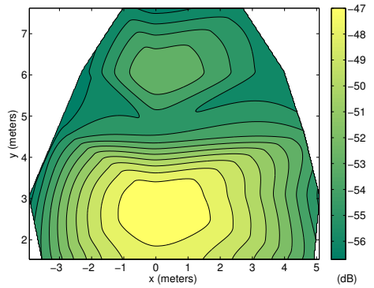



Figure 11: Study the angle of eavesdropping with LNA.

To determine the configuration of the sniffers and the triggers, we conducted an epitomical study using a USRP with two daughterboards attached, one recording at 125 kHz and the other recording at 315 MHz. Our results are depicted in Figure 9 and show that the activation signal of TPS-B lasts approximately 359 ms. The sensors start to transmit 530 ms after the beginning of the activation signal, and the data takes 15 ms to transmit. This means, that to trigger a car traveling at 60 km/h, the trigger should have a transmission range of at least 6 meters. Since a sniffer can eavesdrop up to 9.1 meters, it suffices to place the sniffer right next to the trigger. Additional sniffers could be placed down the road to capture packets of cars traveling at higher speeds.

To determine the feasibility of this approach, we have conducted a roadside experiment using the ATEQ VT55 which has a transmission range of 0.5 meters. We were able to activate and extract the ID of a targeted TPMS sensor moving at the speed of 35 km/h using one sniffer. We note that ATEQ VT55 was deliberately designed with short transmission range to avoid activating multiple cars in the dealership. With a different radio frontend, such as using a matching antenna for 125 kHz, one can increase the transmission range of the trigger easily and enable capturing packets from cars at higher speeds.

**Comparison between tracking via TPMS and Automatic Number Plate Reading.** Automatic Number Plate Reading (ANPR) technologies have been proposed to track automobiles and leverage License Plate Capture Cameras (LPCC) to recognize license plate numbers. Due to the difference between underlying technolo-

| $d$ (m) | $\varphi_1$ (°) | $\varphi_2$(°) | $r_{max}$ (m) |
|---|---|---|---|
| 1.5 | 72.8 | 66.8 | 8.5 |
| 3.0 | 59.1 | 52.4 | 9.1 |
| 4.5 | 45.3 | 31.8 | 7.5 |
| 6.0 | 33.1 | 20.7 | 6.3 |
| 7.5 | 19.6 | 7.7 | 3.8 |

Figure 8: The eavesdropping angles and ranges when the car is traveling at various trajectories.

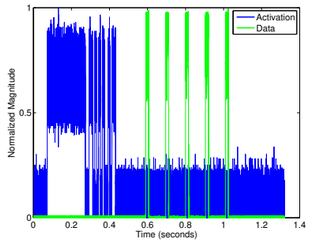



Figure 9: Time series of activation and data signals.

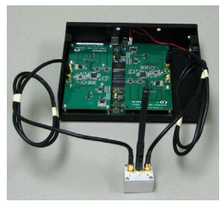


Figure 10: Frequency mixer and USRP with two daughterboards are used to transmit data packets at 315/433 MHz.

gies, TPMS and ANPR systems exhibit different characteristics. First, ANPR allows for more direct linkage to individuals through law enforcement databases. ANPR requires, however, line of sight (LOS) and its accuracy can be affected by weather conditions (e.g. light or humidity) or the dirt on the plate. In an ideal condition with excellent modern systems, the read rate for license plates is approximately 90% [25]. A good quality ANPR camera can recognize number plates at 10 meters [5]. On the contrary, the ability to eavesdrop on the RF transmission of TPMS packets does not depend on illumination or LOS. The probability of identifying the sensor ID is around 99% when the eavesdropper is placed 2.5 meters away from the lane marks. Second, the LOS requirement forces the ANPR to be installed in visible locations. Thus, a motivated driver can take alternative routes or remove/cover the license plates to avoid being detected. In comparison, the use of TPMS is harder to circumvent, and the ability to eavesdrop without LOS could lead to more pervasive automobile tracking. Although swapping or hiding license plates requires less technical sophistication, it also imposes much higher legal risks than deactivating TPMS units.

## 5 Feasibility of Packet Spoofing

Being able to eavesdrop on TPMS communication from a distance allows us to further explore the feasibility of inserting forged data into safety-critical in-vehicle systems. Such a threat presents potentially even greater risks than the tracking risks discussed so far. While the TPMS is not yet a highly safety-critical system, we experimented with spoofing attacks to understand: (1) whether the receiver sensitivity of an in-car radio is high enough to allow spoofing from outside the vehicle or a neighboring vehicle, and (2) security mechanisms and practices in such systems. In particular, we were curious whether the system uses authentication, input validation, or filtering mechanisms to reject suspicious packets.

**The packet spoofing system.** Our live eavesdropper can detect TPMS transmission and decode both ASK modulated TPS-A messages and FSK modulated TPS-B messages in real time. Our packet spoofing system is built on top of our live eavesdropper, as shown in Figure 12. The Packet Generator takes two sets of parameters —*sensor type* and *sensor ID* from the eavesdropper; *temperature*, *pressure*, and *status flags* from users—and generates a properly formulated message. It then modulates the message at baseband (using ASK or FSK) while inserting the proper preamble. Finally, the rogue sensor packets are upconverted and transmitted (either continuously or just once) at the desired frequency (315/433 MHz) using a customized GNU radio python script. We note that once the sensor ID and sensor type are captured we can create and repeatedly transmit the forged message at a pre-defined period.

At the time of our experimentation, there were no USRP daughterboards available that were capable of transmitting at 315/433 MHz. So, we used a frequency mixing approach where we leveraged two XCVR2450 daughterboards and a frequency mixer (mini-circuits ZLW11H) as depicted in Fig.10. By transmitting a tone out of one XCVR2450 into the LO port of the mixer, we were able to mix down the spoofed packet from the other XCVR2450 to the appropriate frequency. For 315 MHz, we used a tone at 5.0 GHz and the spoofed packet at 5.315 GHz.[1]

To validate our system, we decoded spoofed packets with the TPMS trigger tool. Figure 13 shows a screen snapshot of the ATEQ VT55 after receiving a spoofed packet with a sensor ID of "DEADBEEF" and a tire pressure of 0 PSI. This testing also allowed us to understand the meaning of remaining status flags in the protocol.

### 5.1 Exploring Vehicle Security

We next used this setup to send various forged packets to a car using TPS-A sensors (belonging to one of the

[1]For 433 MHz, the spoofed packet was transmitted at 5.433 GHz. We have also successfully conducted the experiment using two RFX-1800 daughterboards, whose operational frequencies are from 1.5 GHz to 2.1 GHz.



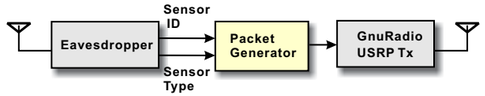



Figure 12: Block chart of the packet spoofing system.

authors) at a rate of 40 packets per second. We made the following observations.

**No authentication.** The vehicle ECU ignores packets with a sensor ID that does not match one of the known IDs of its tires, but appears to accept all other packets. For example, we transmitted forged packets with the ID of the left front tire and a pressure of 0 PSI and found 0 PSI immediately reflected on the dashboard tire pressure display. By transmitting messages with the alert bit set we were able to immediately illuminate the *low-pressure warning light*[2], and with about 2 seconds delay the *vehicle's general-information warning light*, as shown in Figure 14.

**No input validation and weak filtering.** We forged packets at a rate of 40 packets per second. Neither this increased rate, nor the occasional different reports by the real tire pressure sensor seemed to raise any suspicion in the ECU or any alert that something was wrong. The dashboard simply displayed the spoofed tire pressure. We next transmitted two packets with very different pressure values alternately at a rate of 40 packets per second. The dashboard display appeared to randomly alternate between these values. Similarly, when alternating between packets with and without the alert flag, we observed the warning lights switched on and off at non-deterministic time intervals. Occasionally, the display seemed to freeze on one value. These observations suggest that TPMS ECU employs trivial filtering mechanisms which can be easily confused by spoofed packets.

Interestingly, the illumination of the low-pressure warning light depends only on the alert bit—the light turns on even if the rest of the message reports a normal tire pressure of 32 PSI! This further illustrates that the ECU does not appear to use any input validation.

**Large range of attacks.** We first investigated the effectiveness of packet spoofing when vehicles are stationary. We measured the attack range when the packet spoofing system was angled towards the head of the car, and we observed a packet spoofing range of 38 meters. For the purpose of proving the concept, we only used low-cost antennas and radio devices in our experiments. We believe that the range of packet spoofing can be greatly expanded by applying amplifiers, high-gain antennas, or antenna arrays.

[2]To discover this bit we had to deflate one tire and observe the tire pressure sensors response. Simply setting a low pressure bit or reporting low pressure values did not trigger any alert in the vehicle.

**Feasibility of Inter-Vehicle Spoofing.** We deployed the attacks against willing participants on highway I-26 to determine if they are viable at high speeds. Two cars owned by the authors were involved in the experiment. The victim car had TPS-A sensors installed and the attacker's car was equipped with our packet spoofing system. Throughout our experiment, we transmitted alert packets using the front-left-tire ID of the target car, while the victim car was traveling to the right of the attacker's car. We observed that the attacker was able to trigger both the low-pressure warning light and the car's central-warning light on the victim's car when traveling at 55 km/h and 110 km/h, respectively. Additionally, the low-pressure-warning light illuminated immediately after the attacker entered the packet spoofing range.

### 5.2 Exploring the Logic of ECU Filtering

Forging a TPMS packet and transmitting it at a high rate of 40 packets per second was useful to validate packet spoofing attacks and to gauge the spoofing range. Beyond this, though, it was unclear whether there were further vulnerabilities in the ECU logic. To characterize the logic of the ECU filtering mechanisms, we designed a variety of spoofing attacks. The key questions to be answered include: (1) what is the minimum requirement to trigger the TPMS warning light once, (2) what is the minimum requirement to keep the TPMS warning light on for an extended amount of time, and (3) can we permanently illuminate any warning light even after stopping the spoofing attack?

So far, we have observed two levels of warning lights: TPMS Low-Pressure Warning light (TPMS-LPW) and the vehicle's general-information warning light illustrating 'Check Tire Pressure'. In this section, we explored the logic of filtering strategies related to the TPMS-LPW light in detail. The logic controlling the vehicle's general-information warning light can be explored in a similar manner.

#### 5.2.1 Triggering the TPMS-LPW Light

To understand the minimum requirement of triggering the TPMS-LPW light, we started with transmitting one spoofed packet with the rear-left-tire ID and eavesdropping the entire transmission. We observed that (1) one spoofed packet was not sufficient to trigger the TPMS-LPW light; and (2) as a response to this packet, the TPMS ECU immediately sent two activation signals through the antenna mounted close to the rear left tire, causing the rear left sensor to transmit eight packets. Hence, although a single spoofed packet does not cause the ECU to display any warning, it does open a vulnerability to battery drain attacks.


Figure 13: The TPMS trigger tool displays the spoofed packet with the sensor ID "DEADBEEF". We crossed out the brand of TP sensors to avoid legal issues.

(a) (b)

Figure 14: Dash panel snapshots: (a) the tire pressure of left front tire displayed as 0 PSI and the low tire pressure warning light was illuminated immediately after sending spoofed alert packets with 0 PSI; (b) the car computer turned on the general warning light around 2 seconds after keeping sending spoofed packets.

Next, we gradually increased the number of spoofed packets, and we found that transmitting four spoofed packets in one second suffices to illuminate the TPMS-LPW light. Additionally, we found that those four spoofed packets have to be at least 225 ms apart, otherwise multiple spoofed packets will be counted as one. When the interval between two consecutive spoofed packets is larger than 4 seconds or so, the TPMS-LPW no longer illuminates. This indicates that TPMS adopts two detection windows with sizes of 240 ms (a packet lasts for 15 ms) and 4 seconds. A 240-ms window is considered positive for low tire pressure if at least one low-pressure packet has been received in that window regardless of the presence of numerous normal packets. Four 240-ms windows need to be positive to illuminate the TPMS-LPW light. However, the counter for positive 240-ms windows will be reset if no low-pressure packet is received within a 4-s window.

Although the TPMS ECU does use a counting threshold and window-based detection strategies, they are designed to cope with occasionally corrupted packets in a benign situation and are unable to deal with malicious spoofing. Surprisingly, although the TPMS ECU does receive eight normal packets transmitted by sensors as a response to its queries, it still concludes the low-tire-pressure status based on one forged packet, ignoring the majority of normal packets!

#### 5.2.2 Repeatedly Triggering the TPMS-LPW Light

The TPMS-LPW light turns off a few seconds if only four forged packets are received. To understand how to sustain the warning light, we repeatedly transmitted spoofed packets and increased the spoofing period gradually. The TPMS-LPW light remained illuminated when we transmitted the low-pressure packet at a rate higher than one packet per 240 ms, e.g., one packet per detection window. Spoofing at a rate between one packet per 240 ms to 4 seconds caused the TPMS-LPW light to toggle between on and off. However, spoofing at a rate slower than 4 seconds could not activate the TPMS-LPW light, which confirmed our prior experiment results. Figure 15 depicts the measured TPMS-LPW light on-durations and off-durations when the spoofing periods increased from 44 ms to 4 seconds.

As we increased the spoofing period, the TPMS-LPW light remained on for about 6 seconds on average, but the TPMS-LPW light stayed off for an incrementing amount of time which was proportional to the spoofing period. Therefore, it is very likely that the TPMS-ECU adopts a timer to control the minimum on-duration and the off-duration of TPMS-LPW light can be modeled as $t_{off} = 3.5x + 4$, where $x$ is the spoofing period. The off-duration includes the amount of time to observe four low-pressure forged messages plus the minimum waiting duration for the TPMS-ECU to remain off, e.g., 4 seconds. In fact, this confirms our observation that there is a waiting period of approximately 4 seconds before the TPMS warning light was first illuminated.

#### 5.2.3 Beyond Triggering the TPMS-LPW Light

Our previous spoofing attacks demonstrated that we can produce false TPMS-LPW warnings. In fact, transmitting forged packets at a rate higher than one packet per second also triggered the vehicle's general-information warning light illustrating 'Check Tire Pressure'. Depending on the spoofing period, the gap between the illumination of the TPMS-LPW light and the vehicle's general-information warning light varied between a few seconds to 130 seconds — and the TPMS-LPW light remained illuminated afterwards.

Throughout our experiments, we typically exposed the car to spoofed packets for a duration of several minutes at a time. While the TPMS-LPW light usually disappeared about 6 seconds after stopping spoofed message transmissions, we were once unable to reset the light even by turning off and restarting the ignition. It did, however, reset after about 10 minutes of driving.

To our surprise, at the end of only two days of sporadic experiments involving triggering the TPMS warning on and off, we managed to crash the TPMS ECU and



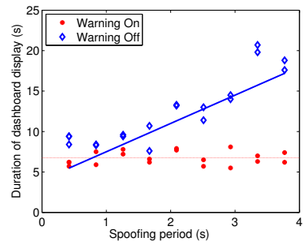
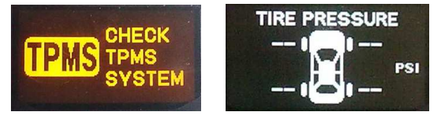



Figure 15: TPMS low-pressure warning light on and off duration vs. spoofing periods.

completely disabled the service. The vehicle's general-information warning light illustrating 'Check TPMS System' was activated and no tire pressure information was displayed on the dashboard, as shown in Figure 16. We attempted to reset the system by sending good packets, restarting the car, driving on the highway for hours, and unplugging the car battery. None of these endeavors were successful. Eventually, a visit to a dealership recovered the system at the cost of replacing the TPMS ECU. This incident suggests that it may be feasible to crash the entire TPMS and the degree of such an attack can be so severe that the owner has no option but to seek the services of a dealership. We note that one can easily explore the logic of a vehicle's general-information warning light using similar methods for TPMS-LPW light. We did not pursue further analysis due to the prohibitive cost of repairing the TPMS ECU.

### 5.3 Lessons Learned

The successful implementation of a series of spoofing attacks revealed that the ECU relies on sensor IDs to filter packets, and the implemented filter mechanisms are not effective in rejecting packets with conflicting information or abnormal packets transmitted at extremely high rates. In fact, the current filer mechanisms introduce security risks. For instance, the TPMS ECU will trigger the sensors to transmit several packets after receiving one spoofed message. Those packets, however, are not leveraged to detect conflicts and instead can be exploited to launch battery drain attacks. In summary, the absence of authentication mechanisms and weak filter mechanisms open many loopholes for adversaries to explore for more 'creative' attacks. Furthermore, despite the unavailability of a radio frontend that can transmit at 315/433 MHz, we managed to launch the spoofing attack using a frequency mixer. This result is both encouraging and alarming since it shows that an adversary can spoof packets even without easy access to transceivers that operate at the target frequency band.




(a) (b)

Figure 16: Dash panel snapshots indicating the TPMS system error (this error cannot be reset without the help of a dealership): (a) the vehicle's general-information warning light; (b) tire pressure readings are no longer displayed as a result of system function errors.

## 6 Protecting TPMS Systems from Attacks

There are several steps that can improve the TPMS dependability and security. Some of the problems arise from poor system design, while other issues are tied to the lack of cryptographic mechanisms.

### 6.1 Reliable Software Design

The first recommendation that we make is that software running on TPMS should follow basic reliable software design practices. In particular, we have observed that it was possible to convince the TPMS control unit to display readings that were clearly impossible. For example, the TPMS packet format includes a field for tire pressure as well as a separate field for warning flags related to tire pressure. Unfortunately, the relationship between these fields were not checked by the TPMS ECU when processing communications from the sensors. As noted earlier, we were able to send a packet containing a legitimate tire pressure value while also containing a low tire pressure warning flag. The result was that the driver's display indicated that the tire had low pressure even though its pressure was normal. A straight forward fix for this problem (and other similar problems) would be to update the software on the TPMS control unit to perform consistency checks between the values in the data fields and the warning flags. Similarly, when launching message spoofing attacks, although the control unit does query sensors to confirm the low pressure, it neglects the legitimate packet responses completely. The control unit could have employed some detection mechanism to, at least, raise an alarm when detecting frequent conflicting information, or have enforced some majority logic operations to filter out suspicious transmissions.

### 6.2 Improving Data Packet Format

One fundamental reason that eavesdropping and spoofing attacks are feasible in TPMS systems is that packets are transmitted in plaintext. To prevent these attacks, a

first line of defense is to encrypt TPM packets[3]. The basic packet format in a TPMS system included a sensor ID field, fields for temperature and tire pressure, fields for various warning flags, and a checksum. Unfortunately, the current packet format used is ill-suited for proper encryption, since naively encrypting the current packet format would still support dictionary-based cryptanalysis as well as replay attacks against the system. For this reason, we recommend that an additional sequence number field be added to the packet to ensure freshness of a packet. Further, requiring that the sequence number field be incremented during each transmission would ensure that subsequent encrypted packets from the same source become indistinguishable, thereby making eavesdropping and cryptanalysis significantly harder. We also recommend that an additional cryptographic checksum (e.g. a message authentication code) be placed prior to the CRC checksum to prevent message forgery.

Such a change in the payload would require that TPMS sensors have a small amount memory in order to store cryptographic keys, as well as the ability to perform encryption. An obvious concern is the selection of cryptographic algorithms that are sufficiently light-weight to be implemented on the simple processor within a TPMS sensor, yet also resistant to cryptanalysis. A secondary concern is the installation of cryptographic keys. We envision that the sensors within a tire would be have keys pre-installed, and that the corresponding keys could be entered into the ECU at the factory, dealership, or a certified garage. Although it is unlikely that encryption and authentication keys would need to be changed, it would be a simple matter to piggy-back a rekeying command on the 125kHz activation signal in a manner that only certified entities could update keys.

### 6.3 Preventing Spoofed Activation

The spoofing of an activation signal forces sensors to emit packets and facilitates tracking and battery drain attacks. Although activation signals are very simple, they can convey a minimal amount of bits. Thus, using a long packet format with encryption and authentication is unsuitable, and instead we suggest that the few bits they can convey be used as a sequencing field, where the sequencing follows a one-way function chain in a manner analogous to one-time signatures. Thus, the ECU would be responsible for maintaining the one-way function chain, and the TPMS sensor would simply *hash* the observed sequence number and compare with the previous sequence number. This would provide a simple means of filtering out false activation signals. We note that other legitimate sources of activation signals are specialized entities, such as dealers and garages, and such entities could access an ECU to acquire the position within the hash chain in order to reset their activation units appropriately to allow them to send valid activation signals.

[3] We note that encrypting the entire message (or at least all fields that are not constant across different cars) is essential as otherwise the ability to read these fields would support a privacy breach.

## 7 Related Work

Wireless devices have become an inseparable part of our social fabric. As such, much effort has been dedicated to analyze the their privacy and security issues. Devices being studied include RFID systems [27, 30, 41], mass-market UbiComp devices [38], household robots [14], and implantable medical devices [21]. Although our work falls in the same category and complements those works, TPMS in automobiles exhibits distinctive features with regard to the radio propagation environment (strong reflection within and off metal car bodies), ease of access by adversaries (cars are left unattended in public), span of usage, a tight linkage to the owners, etc. All these characteristics have motivated this in-depth study on the security and privacy of TPMS.

One related area of research is location privacy in wireless networks, which has attracted much attention since wireless devices are known to present tracking risks through explicit identifiers in protocols or identifiable patterns in waveforms. In the area of WLAN, Brik *et al.* have shown the possibility to identify users by monitoring radiometric signatures [10]. Gruteser *et al.* [19] demonstrated that one can identify a user's location through link- and application-layer information. A common countermeasure against breaching location privacy is to frequently dispose user identity. For instance, Jiang *et al.* [24] proposed a pseudonym scheme where users change MAC addresses each session. Similarly, Greenstein *et al.* [18] have suggested an identifier-free mechanism to protect user identities, whereby users can change addresses for each packet.

In cellular systems, Lee *et al.* have shown that the location information of roaming users can be released to third parties [28], and proposed using the temporary mobile subscriber identifier to cope with the location privacy concern. IPv6 also has privacy concerns caused by the fixed portion of the address [32], and thus the use of periodically varying pseudo-random addresses has been recommended. The use of pseudonyms is not sufficient to prevent automobile tracking since the sensors report tire pressure and temperature readings, which can be used to build a signature of the car. Furthermore, pseudonyms cannot defend against packet spoofing attacks such as we have examined in this paper.

Security and privacy in wireless sensor networks have been studied extensively. Perrig *et al.* [37] have proposed a suite of security protocols to provide data confidentiality and authentication for resource-constrained sensors. Random key predistribution schemes [12] have been proposed to establish pairwise keys between sensors on demand. Those key management schemes cannot work well with TPMS, since sensor networks are concerned with establishing keys among a large number of sensors while the TPMS focuses on establishing keys between four sensors and the ECU only.

Lastly, we note related work on the security of a car's computer system [26]. Their work involved analyzing the computer security within a car by directly mounting a malicious component into a car's internal network via the On Broad Diagnostics (OBD) port (typically under the dash board), and differs from our work in that we were able to remotely affect an automobile's security at distances of 40 meters without entering the car at all.

## 8 Concluding Remarks

Tire Pressure Monitoring Systems (TPMS) are the first in-car wireless network to be integrated into all new cars in the US and will soon be deployed in the EU. This paper has evaluated the privacy and security implications of TPMS by experimentally evaluating two representative tire pressure monitoring systems. Our study revealed several security and privacy concerns. First, we reverse engineered the protocols using the GNU Radio in conjunction with the Universal Software Radio Peripheral (USRP) and found that: (i) the TPMS does not employ any cryptographic mechanisms and (ii) transmits a fixed sensor ID in each packet, which raises the possibility of tracking vehicles through these identifiers. Sensor transmissions can be triggered from roadside stations through an activation signal. We further found that neither the heavy shielding from the metallic car body nor the low-power transmission has reduced the range of eavesdropping sufficiently to reduce eavesdropping concerns. In fact, TPMS packets can be intercepted up to 40 meters from a passing car using the GNU Radio platform with a low-cost, low-noise amplifier. We note that the eavesdropping range could be further increased with directional antennas, for example.

We also found out that current implementations do not appear to follow basic security practices. Messages are not authenticated and the vehicle ECU also does not appear to use input validation. We were able to inject spoofed messages and illuminate the low tire pressure warning lights on a car traveling at highway speeds from another nearby car, and managed to disable the TPMS ECU by leveraging packet spoofing to repeatedly turn on and off warning lights.

Finally, we have recommended security mechanisms that can alleviate the security and privacy concerns presented without unduly complicating the installation of new tires. The recommendations include standard reliable software design practices and basic cryptographic recommendations. We believe that our analysis and recommendations on TPMS can provide guidance towards designing more secure in-car wireless networks.

## References


[1] Ettus Research LLC. http://www.ettus.com/.

[2] GNU radio. http://gnuradio.org.

[3] IEEE 1609: Family of Standards for Wireless Access in Vehicular Environments (WAVE). http://www.standards.its.dot.gov/fact_sheet.asp?f=80.

[4] Portable, solar-powered tag readers could improve traffic management. Available at http://news.rpi.edu/update.do?artcenterkey=1828.

[5] RE-BCC7Y Number plate recognition cameras. http://www.dsecctv.com/Prod_lettura_targhe.htm.

[6] Traffic hackers hit red light. Available at http://www.wired.com/science/discoveries/news/2005/08/68507.

[7] Improving the safety and environmental performance of vehicles. *EUROPA-Press Releases* (23rd May 2008).

[8] ATEQ VT55. http://www.tpms-tool.com/tpms-tool-ateqvt55.php.

[9] BALANIS, C., AND IOANNIDES, P. Introduction to smart antennas. *Synthesis Lectures on Antennas 2*, 1 (2007), 1–175.

[10] BRIK, V., BANERJEE, S., GRUTESER, M., AND OH, S. Wireless device identification with radiometric signatures. In *Proceedings of ACM International Conference on Mobile Computing and Networking (MobiCom)* (2008), ACM, pp. 116–127.

[11] BRZESKA, M., AND CHAKAM, B. RF modelling and characterization of a tyre pressure monitoring system. In *EuCAP 2007: The Second European Conference on Antennas and Propagation* (2007), pp. 1 – 6.

[12] CHAN, H., PERRIG, A., AND SONG, D. Random key predistribution schemes for sensor networks. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy* (2003), IEEE Computer Society, p. 197.

[13] COLE, G., AND SHERMAN, A. Lightweight materials for automotive applications. *Materials Characterization 35* (1995), 3–9.

[14] DENNING, T., MATUSZEK, C., KOSCHER, K., SMITH, J. R., AND KOHNO, T. A spotlight on security and privacy risks with future household robots: attacks and lessons. In *Ubicomp '09: Proceedings of the 11th international conference on Ubiquitous computing* (2009), pp. 105–114.

[15] FERESIDIS, A., AND VARDAXOGLOU, J. High gain planar antenna using optimised partially reflective surfaces. In *IEEE Proceedings on Microwaves, Antennas and Propagation* (2001), vol. 148, pp. 345 – 350.

[16] FREDRIKSSON, L., AND AB, K. Bluetooth in automotive applications. *http://www.kvaser.com/can/info/files/bluetooth-in-automotive-appl.pdf*.

[17] GOVINDJEE, S. Firestone tire failure analysis, 2001.

[18] GREENSTEIN, B., MCCOY, D., PANG, J., KOHNO, T., SESHAN, S., AND WETHERALL, D. Improving wireless privacy with an identifier-free link layer protocol. In *Proceeding of Mobile systems, applications, and services (MobiSys)* (2008), ACM, pp. 40–53.

[19] GRUTESER, M., AND GRUNWALD, D. A methodological assessment of location privacy risks in wireless hotspot networks. In *Security in Pervasive Computing, First International Conference* (2003), pp. 10–24.

[20] GRUTESER, M., AND GRUNWALD, D. Enhancing location privacy in wireless lan through disposable interface identifiers: a quantitative analysis. *ACM Mobile Networks and Applications (MONET) 10*, 3 (2005), 315–325.

[21] HALPERIN, D., HEYDT-BENJAMIN, T. S., RANSFORD, B., CLARK, S. S., DEFEND, B., MORGAN, W., FU, K., KOHNO, T., AND MAISEL, W. H. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Proceedings of IEEE Symposium on Security and Privacy* (2008), IEEE Computer Society, pp. 129–142.

[22] IEEE 802.11P. IEEE draft standard for information technology -telecommunications and information exchange between systems. http://www.ieee802.org/11/.

[23] ITALY. *http://aglobalworld.com/international-countries/Europe/Italy.php*.

[24] JIANG, T., WANG, H. J., AND HU, Y.-C. Preserving location privacy in wireless lans. In *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services* (2007), ACM, pp. 246–257.

[25] KEILTHY, L. Measuring ANPR System Performance. *Parking Trend International* (2008).

[26] KOSCHER, K., CZESKIS, A., ROESNER, F., PATEL, S., KOHNO, T., CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., AND SAVAGE, S. Experimental security analysis of a modern automobile. In *Proceedings of IEEE Symposium on Security and Privacy in Oakland* (May 2010).

[27] KOSCHER, K., JUELS, A., BRAJKOVIC, V., AND KOHNO, T. EPC RFID tag security weaknesses and defenses: passport cards, enhanced drivers licenses, and beyond. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), pp. 33–42.

[28] LEE, C.-H., HWANG, M.-S., AND YANG, W.-P. Enhanced privacy and authentication for the global system for mobile communications. *Wireless Networks 5*, 4 (1999), 231–243.

[29] LIEDTKE, F. Computer simulation of an automatic classification procedure for digitally modulated communication signals with unknown parameters. *Signal Processing 6* (1984), 311–323.

[30] MOLNAR, D., AND WAGNER, D. Privacy and security in library RFID: issues, practices, and architectures. In *Proceedings of Computer and communications security* (2004), ACM Press, pp. 210–219.

[31] MURPHY, N. A short trip on the can bus. *Embedded System Programming* (2003).

[32] NARTEN, T., DRAVES, R., AND KRISHNAN, S. RFC 4941 - privacy extensions for stateless address autoconfiguration in IPv6, Sept 2007.

[33] NUSSER, R., AND PELZ, R. Bluetooth-based wireless connectivity in an automotive environment. *Vehicular Technology Conference 4* (2000), 1935 – 1942.

[34] OF TRANSPORTATION, B. Number of vehicles and vehicle classification, 2007.

[35] OF TRANSPORTATION NATIONAL HIGHWAY, D., AND ADMINISTRATION, T. S. 49 cfr parts 571 and 585 federal motor vehicle safety standards; tire pressure monitoring systems; controls and displays; final rule. *http://www.tireindustry.org/pdf/TPMS_FinalRule_v3.pdf*.

[36] PAPADIMITRATOS, P., BUTTYAN, L., HOLCZER, T., SCHOCH, E., FREUDIGER J., RAYA, M., MA, Z., KARGL, F., KUNG, A., AND HUBAUX, J.-P. Secure Vehicular Communication Systems: Design and Architecture. *IEEE Communcations Magazine 46*, 11 (November 2008), 100–109.

[37] PERRIG, A., SZEWCZYK, R., WEN, V., CULLER, D., AND TYGAR, J. D. Spins: security protocols for sensor networks. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking* (2001), ACM, pp. 189–199.

[38] SAPONAS, T. S., LESTER, J., HARTUNG, C., AGARWAL, S., AND KOHNO, T. Devices that tell on you: privacy trends in consumer ubiquitous computing. In *Proceedings of USENIX Security Symposium* (2007), USENIX Association, pp. 1–16.

[39] SONG, H., COLBURN, J., HSU, H., AND WIESE, R. Development of reduced order model for modeling performance of tire pressure monitoring system. In *IEEE 64th Vehicular Technology Conference* (2006), pp. 1 – 5.

[40] VELUPILLAI, S., AND GUVENC, L. Tire pressure monitoring. *IEEE Control Systems Magazine 27* (2007), 22–25.

[41] WEIS, S. A., SARMA, S. E., RIVEST, R. L., AND ENGELS, D. W. Security and privacy aspects of low-cost radio frequency identification systems. In *Security in Pervasive Computing* (2004), vol. 2802 of *Lecture Notes in Computer Science*, pp. 201–212.

[42] YEH, P., STARK, W., AND ZUMMO, S. Performance analysis of wireless networks with directional antennas. *IEEE Transactions on Vehicular Technology 57*, 5 (2008), 3187–3199.

# VEX: Vetting Browser Extensions For Security Vulnerabilities

Sruthi Bandhakavi Samuel T. King P. Madhusudan Marianne Winslett
*University of Illinois at Urbana Champaign*
{sbandha2,kingst,madhu,winslett}@*illinois.edu*

## Abstract

The browser has become the *de facto* platform for everyday computation. Among the many potential attacks that target or exploit browsers, vulnerabilities in browser extensions have received relatively little attention. Currently, extensions are vetted by manual inspection, which does not scale well and is subject to human error.

In this paper, we present VEX, a framework for highlighting potential security vulnerabilities in browser extensions by applying static information-flow analysis to the JavaScript code used to implement extensions. We describe several patterns of flows as well as unsafe programming practices that may lead to privilege escalations in Firefox extensions. VEX analyzes Firefox extensions for such flow patterns using high-precision, context-sensitive, flow-sensitive static analysis. We analyze thousands of browser extensions, and VEX finds six exploitable vulnerabilities, three of which were previously unknown. VEX also finds hundreds of examples of bad programming practices that may lead to security vulnerabilities. We show that compared to current Mozilla extension review tools, VEX greatly reduces the human burden for manually vetting extensions when looking for key types of dangerous flows.

## 1 Introduction

Driving the Internet revolution is the modern web browser, which has evolved from a relatively simple client application designed to display static data into a complex networked operating system tasked with managing many facets of a user's on-line experience. To help meet the varied needs of a broad user population, *browser extensions* expand the functionality of browsers by interposing on and interacting with browser-level events and data. Some extensions are simple and make only small changes to the appearance of web pages or the browser itself. Other extensions provide more sophisticated functionality, such as NOSCRIPT that provides fine-grained control over JavaScript execution [20], or GREASEMONKEY that provides a full-blown programming environment for scripting browser behavior [6]. These are just a few of the thousands of extensions currently available for Firefox, the second most popular browser today[1].

Extensions written with benign intent can have subtle vulnerabilities that expose the user to a disastrous attack from the web, often just by viewing a web page. Firefox extensions run with full browser privileges, so attackers can potentially exploit extension weaknesses to take over the browser, steal cookies or protected passwords, compromise confidential information, or even hijack the host system, without revealing their actions to the user. Unfortunately, tens of extension vulnerabilities have been discovered in the last few years, and capable attacks against buggy extensions have already been demonstrated [23].

To help reduce the attack surface for extensions, Mozilla provides a set of security primitives to extension developers. However, these security primitives are discretionary, and can be difficult to understand and use correctly. For example, Firefox provides an `evalInSandbox (text, sandbox)` function that returns the result of evaluating the `text` string under the restricted privileges associated with the environment `sandbox`. Using `evalInSandbox` correctly requires developers to test the result of a call to `evalInSandbox` with the non-traditional "`===`" rather than "`==`", as the "`==`" operation may invoke unsafe code as a side effect (See `http://developer.mozilla.org/En/Components.utils.evalInSandbox` for details).

Current approaches from the research community propose dynamic techniques for improving the security of extensions. The SABRE system tracks tainted JavaScript

[1] Firefox now surpasses Internet Explorer in W3schools traffic (`www.w3schools.com/browsers/browsers_stats.asp`), arguably due to the popularity of Firefox extensions.

objects to prevent extensions from accessing sensitive information unsafely [9]. Although SABRE can prevent potentially malicious flows from both exploited extensions and from malicious extensions, SABRE adds overhead to all JavaScript execution within the browser, adding 6.1x overhead for the SunSpider benchmark and 2.36x overhead for the V8 JavaScript benchmark. Furthermore, SABRE's dynamic nature pushes security violation notification to users who are unable to determine if a particular flow is malicious or benign. The Google Chrome Extension System revisits the overall extension API to make it easier for the browser to enforce least privilege and strong isolation on extensions [4]. Their system works by partitioning the full set of extension functionality into different protection domains, and sand-boxing extensions to prevent them from obtaining more privileges than needed. Although this system is likely to limit the damage from some extension attacks, it does little to prevent the vulnerabilities themselves.

In this paper, we propose VEX, a system for finding vulnerabilities in browser extensions using static information-flow analysis. Many vulnerabilities translate to certain types of *explicit information flows* from injectable sources to executable sinks. For extensions written with benign intent, most attacks involve the attacker injecting JavaScript into a data item that is subsequently executed by the extension under full browser privileges. We identify key flows of this nature that can lead to security vulnerabilities, and we analyze for these flows statically using high-precision static analysis that is both path-sensitive and context-sensitive, to minimize the number of false positive suspect flows. VEX uses precise summaries to analyze code, and has special features to handle the quirks of JavaScript (e.g., VEX does a constant string analysis for expressions that flow into the `eval` statement). Because VEX uses static analysis, we avoid the runtime overhead induced by dynamic approaches.

Determining whether extensions are malicious or harbor security vulnerabilities is a hard problem. Extensions are typically complex artifacts that interact with the browser in subtle and hard to understand ways. For example, the ADBLOCK PLUS extension performs the seemingly simple task of filtering out ads based on a list of ad servers. However, the ADBLOCK PLUS implementation consists of over 11K lines of JavaScript code. Similarly, the NOSCRIPT extension provides fine-grained control over which domains are allowed to execute JavaScript and basic cross-site scripting protection. The NOSCRIPT extension implementation consists of over 19K lines of JavaScript code. Also, ADBLOCK PLUS had 30 releases in 1/1/06–11/20/09, and NOSCRIPT had 38 releases just in 1/1/09–11/20/09. While Mozilla uses volunteers to vet each new extension and revision before posting it on their official list of approved Firefox extensions, examining an extension to find a vulnerability requires a detailed understanding of the code to reason about anything beyond the most basic type of information flow. Thus tools to help vet browser extensions can be very useful for improving the security of extensions.

We show that VEX can catch several known vulnerabilities, such as a vulnerability in the FIZZLE extension [8], and also find new problems, including exploitable vulnerabilities in BEATNIK and WIKIPEDIA TOOLBAR. In particular, VEX reported a previously unknown vulnerability in WIKIPEDIA TOOLBAR that could lead to an attack, and that resulted in the report CVE-2009-4127. We reported this vulnerability to the WIKIPEDIA TOOLBAR developers, who fixed the extension. We also show that VEX can help to find the use of unsafe programming practices, such as misuse of `evalInSandbox`, that can result from subtle information flows.

The remainder of the paper is organized as follows. Section 2 describes the threat model and the assumptions under which we analyze the browser extensions. Section 3 provides background material on the architecture of Firefox and the nature of certain key undesirable information flows in its extensions. Section 4 describes our static analysis and the various design choices we made to build VEX. Section 5 lists and describes our results. Section 6 surveys related work, and Section 7 concludes the paper.

## 2 Threat model, assumptions, and usage model

In this paper, we focus on finding security vulnerabilities in buggy browser extensions. We do not try to identify malicious extensions, bugs in the browser itself, or bugs in other browser extensibility mechanisms, such as plug-ins. We assume that the developer is neither malicious nor trying to obfuscate extension functionality, but we assume the developer could write incorrect code that contains vulnerabilities.

We use two attack models. First, we consider attacks that originate from web sites, and we assume the attacker can send arbitrary HTML and JavaScript to the user's browser. We focus on attacks where this untrusted data can lead to code injection or privilege escalation through buggy extensions. In the second attack model, we consider some web sites as trusted. For example, if an extension gleans information from Facebook, we assume that the Facebook code will *not* include arbitrary HTML and JavaScript, but only well formatted and trusted data.

According to the Mozilla developer site, Mozilla has a team of volunteers who help vet extensions manually.



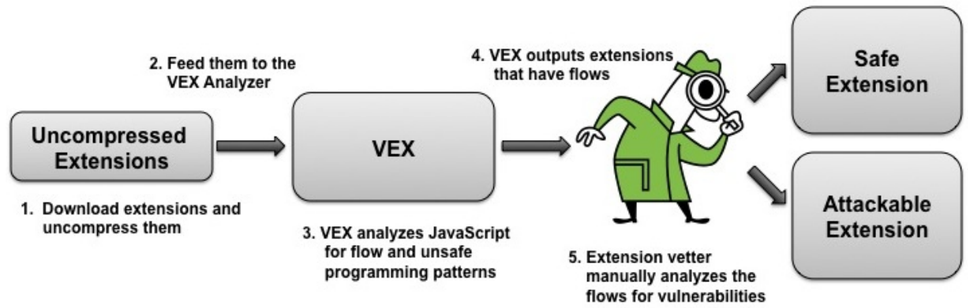



Figure 1: The overall analysis process of VEX.

They run new and updated extensions isolated in a virtual machine to test the user experience. The editors also use a validation tool, which uses *grep* to look for key indicators of bugs. Many of the patterns they search for involve interactions between extensions and web pages, and they use their understanding of these patterns to help guide their inspection of the code. Our goal is to help automate this process, so that analysts can quickly hone in on particular snippets of code that are likely to contain security vulnerabilities. Figure 1 shows our overall work flow for using VEX.

## 3 Background

### 3.1 Mozilla privilege levels

Firefox has two privilege levels: *page*, for the web page displayed in the browser's content pane; and *chrome*, for elements belonging to Firefox and its extensions, i.e., everything surrounding the content pane. Page privileges are more restrictive than chrome privileges. For example, a page loaded from site $x$ cannot access content from sites other than $x$. General Firefox code runs with full chrome privileges, which give it access to all browser states and events, OS resources like the file system and network, and all web pages. Firefox provides the extensions with full chrome privileges by exposing a special API called the XPCOM Components to extension JavaScript, thereby allowing the extensions to have access to all the resources Firefox can access.

Extensions can often access objects that run with page privileges and interact with page content, as well as objects that run with full chrome privileges. Extensions can also include user interface components via a *chrome document*, which also runs with full chrome privileges. For example, the object `window` refers to the chrome window and the object `window.content` refers to the content window. To access the `document` object referring to the content (i.e., the user page), the extension has to access the `document` property of the content window, i.e., `window.content.document`.

To make this extension architecture practical, Firefox has APIs for extension code to communicate across protection domains. These interactions are one cause of extension security vulnerabilities. As the Mozilla developer site explains, "One of the most common security issues with extensions is execution of remote code in privileged context. A typical example is an RSS reader extension that would take the content of the RSS feed (HTML code), format it nicely and insert into the extension window. The issue that is commonly overlooked here is that the RSS feed could contain some malicious JavaScript code and it would then execute with the privileges of the extension – meaning that it would get full access to the browser (cookies, history etc) and to user's files" [sic].

### 3.2 Points of attack

Here we discuss key vulnerable points for code injection and privilege escalation attacks against non-malicious extensions: `eval`, `evalInSandbox`, `innerHTML`, and `wrappedJSObject`. We focus on these Firefox features because they are key points of interaction between objects with page and chrome privileges, respectively, and this interaction is a key source of security vulnerabilities, as noted above. Though other avenues of attack are possible, we do not consider them here.

**eval:** The `eval` function call interprets string data as JavaScript, which it executes dynamically. This flexible mechanism can be used to generate JavaScript code dynamically, for example to serialize JSON objects. However, this flexibility can lead to code injection vulnerabilities in extensions. If extensions execute `eval` functions on un-sanitized strings that come from untrusted web pages, the attacker will be able to inject JavaScript code that will run with full chrome privileges.

**InnerHTML:** Each HTML element for a page has an `innerHTML` property that defines the text that occurs between that element's opening and closing tag. Extensions can change the `innerHTML` property to alter existing document object model (DOM) elements, or to add new DOM elements. When an extension modifies the `innerHTML` property, the browser re-parses and processes the new data. Thus, passing specially crafted unsanitized strings (e.g., <img> tags with script in their `onload` attribute) into `innerHTML` modifications can lead to code injection attacks.

**EvalInSandbox:** One way Firefox facilitates communication across protection domains is through the `evalInSandbox` method. This method enables extensions to execute JavaScript in the extension's context with restricted privileges, thus enabling extensions to process untrusted data from web pages safely. The sandbox object is an empty JavaScript object created with restricted privileges. For example, the call `s = Sandbox("http://www.w3.org/")` creates a sandbox `s` where code can execute with page privileges, as though it came from the domain www.w3.org. One can add properties to this object by calling the `evalInSandbox` function, and any attempts to access global scope objects from within `evalInSandbox`, including privileged chrome objects, are denied. `evalInSandbox` complicates extension programming because objects returned from the method call execute in the extension with full chrome privileges. Since methods associated with the object could have been modified within the sandbox, they should not be called in the chrome context. For example, "==" should not be used on these objects as its evaluation calls the `tostring` or `valueOf` method, which could have been modified; instead the non-traditional "===" operator needs to be used.

**wrappedJSObject:** JavaScript objects can be dynamically modified. That means that any web page can modify the properties of the `document` object. For example, a web page can reassign the `getElementById` method to return a malicious script. To prevent this script from being executed by the extension when it calls `window.content.document.getElementById`, Firefox automatically wraps the object so that the `window.content.document` accesses only use the original document object, not the modified one. However, Firefox also provides the `wrappedJSObject` method, which lets the extension access the modified version, even when automatic wrapping is turned on; calling `wrappedJSObject` on a content document is potentially dangerous.

### 3.3 Suspicious flow patterns

In this section we discuss the five source to sink flows that might be vulnerable. Specifically, we track flows from Resource Description Framework (RDF) data (e.g., bookmarks) to `innerHTML`, content document data to `eval`, content document data to `innerHTML`, `evalInSandbox` return objects used improperly by code running with chrome privileges, or `wrappedJSObject` return object used improperly by code running with chrome privileges. These flows do not always result in a vulnerability, and they are by no means an exhaustive list of all possible extension security bugs, but they are the patterns we use in our tool.

RDF is a model for describing hierarchical relationships between browser resources [33]. Extension developers can store persistent extension data in an RDF file, or access browser resources, such as bookmarks, stored in RDF format. RDF data can come from untrusted sources. For example, when a user stores a bookmark, Firefox records the un-sanitized title of the bookmarked page in the RDF file. Extensions that use RDF data need to sanitize it properly if they use it directly in an `innerHTML` statement that modifies an element in a chrome document.

Content document data flowing to `eval` or `innerHTML` can sometimes be exploited. This flow can result in script execution with chrome privileges if specially crafted content from the `window.content.document` object is passed to `eval` or `innerHTML` or an element in the chrome document.

For `evalInSandbox` and `wrappedJSObject`, problems can only result if the return values of these constructs are executed with chrome privileges. For `evalInSandbox` this means comparing return values using == or != from code running with chrome privileges. For `wrappedJSObject`, this means making method calls on returned objects from code running with chrome privileges.

Such flow patterns may occur in only a few of the extensions that use these constructs. According to the Mozilla extension review web page, reviewers have an open-source automatic tool to help with reviews (see `https://addons.mozilla.org/en-US/firefox/pages/validation`), but this tool just greps for strings that indicate dangerous patterns. Afterward, the reviewer must go through the code of each suspect extension to understand the flows and determine which constitute vulnerabilities and which are benign. As this task is difficult, painful, and error-prone, we designed the VEX tool to help extension reviewers vet the flows in extensions automatically, greatly reducing the number of extensions that need manual review.

## 4 Static information flow analysis

We develop a general explicit information flow static analysis tool VEX for JavaScript that computes flows between any source and sink, including the flows described in Section 3.3. While we could develop analysis techniques for a particular source and sink, we prefer a more general technique that will perform the analysis once, and from the results, allow us to search for any source-to-sink flow. This allows VEX to be run in a single pass over thousands of extensions, rather than using separate passes for each target pattern.

To support fine-grained information-flow analysis, VEX tracks the precise dependencies of flows from variables to objects created in the JavaScript extension, using a taint-based analysis. Motivated by the fact that every flow reported needs to be checked manually for attacks, which can take considerable human effort, we aim for an analysis that admits as few false positives as possible (false positives are non-existent flows reported by the tool).

Statically analyzing JavaScript extensions for flows is a non-trivial task. JavaScript extensions have a large number of objects and functions. In addition to the objects defined in the program, the extensions can also access the browser's DOM API and the Firefox Extension API provided by XPCOM components. The objects are also dynamic, in the sense that new object properties can be created dynamically at run-time. Functions are objects in JavaScript, and hence can be created, redefined dynamically, and passed as parameters. The challenge is to accurately keep track of such objects, properties, and the corresponding flows to them.

Our analysis keeps track of an *abstract heap* (AH) that is not *a priori* bounded, and keeps track of the precise heap nodes and field relations and corresponding flows, but ignores the exact primitive values in the heap (like integers). However, we bound *the number of iterations* in computing the least fixed-point, and hence the abstract heap gets bounded implicitly.

The abstract heap transformations for any statement closely mimic a big-step operational semantics for JavaScript, except that primitive values are forgotten, and hence conditionals are not evaluated; we refer the reader to work on operational semantics of JavaScript [27, 18].

Apart from tracking heap structures, the abstract heap also records *explicit-flow* dependencies to heap nodes, and the rules for updating flows naturally depend on the program's semantics. Also, as we talk about in more detail below, there are some aspects of the heap (such as prototype fields) that are not currently supported in our tool. The static analysis itself is flow-sensitive and context-sensitive, and the context-sensitivity is handled using classical function-summary based methods.

The above choices, namely the choice of abstract heaps, and the context-sensitive flow-sensitive analysis, are *design choices* we have made, based on our experiments with extensions for over a year, and were motivated to reduce false positives. However, we have not tried all variants of these choices, and it is possible that other choices (for example, choosing to bound abstract heaps by merging objects created at a program site), may also work well on extensions. However, we do know that *context-sensitivity* is important (in several extensions we manually examined) and further *flow-sensitivity* seems important if the tool is extended to consider sanitization routines as flow-stoppers.

The rest of this section is structured as follows. First we explain our analysis using abstract heaps for a core subset of JavaScript, which does not have statements like `eval`, associative array accesses, calls to Firefox APIs, etc. Subsequently, we describe how we handle the aspects not covered in the core.

### 4.1 Analysis of a core subset of JavaScript

**Core JavaScript:** A core subset of JavaScript is given in Figure 2; this core reflects the aspects of JavaScript described above, but omits certain features (such as `eval`) which we will describe later.

**Abstract Heaps:** Our analysis keeps track of a *one abstract heap* at each program point. This *abstract heap* tracks JavaScript objects and functions and the relationships between them in the form of a graph. Each node in the graph is a heap location generated by the program. Two different nodes, $n_1$ and $n_2$ are connected by an edge labeled $f$, if node $n_1$'s property $f$ *may refer* to $n_2$. To keep track of the actual information flows between different program variables, we also keep track of all the program variables that flow into the nodes in abstract heap. Let $PVar$ be the set of all the program variables in the JavaScript program.

More precisely, an abstract heap $\sigma$ is a tuple ($ns$, $n$,$d$, $fr$, $dm$, $tm$), where:

- $ns$ is a set of heap locations,

EXPRESSIONS ::=

| | |
|---|---|
| $c$ | (CONSTANT) |
| $x$ | (VARIABLE) |
| $x.f$ | (FIELD ACCESS) |
| $x.prot$ | (PROTO ACCESS) |
| $eop\ e$ | (BINARY OP) |
| `this` | (THIS) |
| $\{f_1 : e_1, \ldots, f_n : e_n\}$ | (OBJECT LITERAL) |
| `function` $(p_1, \ldots, p_n)\{S\}$ | (FUNCTION DEF) |
| $f(a_1, \ldots, a_n)$ | (FUNCTION CALL) |
| `new` $f(a_1, \ldots, a_n)$ | (NEW) |

STATEMENTS ::=

| | |
|---|---|
| `skip` | (SKIP) |
| $S_1; S_2$ | (SEQ) |
| `var` $x$ | (VARIABLE DECL.) |
| $x := e$ | (ASSIGN 1) |
| $x.f := e$ | (ASSIGN 2) |
| `if` $e$ `then` $S_1$ `else` $S_2$ | (CONDITIONAL) |
| `while` $e$ `do` $S$ `od` | (WHILE) |
| `return` $e$ | (RETURN) |

Figure 2: Core JavaScript syntax.

- $n \in (ns \cup \{\bot\})$ represents the *current node*, and is either a node in the heap or the symbol $\bot$,
- $d \subseteq PVar$ represents the subset of program variables that flow in to the current node $n$,
- $fr \subseteq ns \times PVar \times (ns \cup \{\bot\})$ encodes the pointers representing properties (fields). A triple $(n_1, f, n_2) \in fr$ means that the property $f$ of the object $n_1$ *may be* located at $n_2$.
- $dm \subseteq ns \times PVar$ is a relation that denotes a *dependency map*. A pair $(n_1, x) \in dm$ denotes that the program variable $x$ flows into the node $n_1$.
- $tm : ns \times ns$ is a "this-map" relation, which is actually the relation of a function. A pair $(n_1, n_2) \in tm$ means that the scope of $n_1$ is $n_2$.

**Notation:** The relation $tm$ will always be a function; we define formally the function $tm : ns \rightarrow ns$ as $tm(n) = n'$, where $(n, n') \in tm$. Let $dm : ns \rightarrow 2^{PVar}$ be the function that corresponds to the relation $dm$, $dm(n) = \{x | (n, x) \in dm\}$, i.e. the set of all the program variables that flow into the node $n$.

**The Analysis:** We now describe our analysis for the core subset of JavaScript. VEX handles functions and objects by creating a node for every object or function and their properties. Relationships between various nodes are accurately generated and tracked in the analysis. JavaScript uses prototype-based inheritance; however, our analysis does not track prototypes. Instead, a new property insertion into the prototype field of an object is treated as if the property is being inserted into the object itself. We found that this is sufficient in case of JavaScript extensions as the inheritance chain is not deep in most cases. VEX keeps track of the accurate scope information using the this-map.

Our analysis consists of a set of rules for generating abstract heaps at program points, and is defined by essentially capturing the effect of statements on the abstract heap. These rules follow a big-step operational semantics adapted to work on the abstracted heap.

The big step operational semantics on abstract heaps is defined as a relation , $(Prog, \sigma) \Downarrow \sigma'$, where `Prog` is an program expression or statement and $\sigma$ and $\sigma'$ are abstract heaps. Such a relation intuitively means that $\sigma'$ is the heap obtained from the complete evaluation of `Prog` starting from the heap $\sigma$. This resulting heap, in every iteration, will be *merged* with the current heap after the program, conservatively taking the union of dependencies.

We now define this relation for expressions and statements.

**Notation:** For any abstract heap $\sigma$, let $\sigma = (ns_\sigma, n_\sigma, d_\sigma, fr_\sigma, dm_\sigma, tm_\sigma)$. In other words, $n_\sigma$ refers to the second component of $\sigma$, etc. The function $fresh()$ creates a new heap location. A special node $n_G$ represents the global heap, which consists of the objects like `Object`, `Array`, etc.

**Evaluating expressions:**
Figure 3 gives the rules for evaluating expressions in the program.

Rule (CONSTANT) evaluates to a $\bot$ node with empty dependencies. Rule (THIS) extracts the scope of the current node. The next five rules describe the variable and field access expressions.

In case of a variable access, the existence property $x$ is checked in the current scope (represented by $n_\sigma$(rule (VAR))), and returned if it exists. If it is not in the current scope, then the global node (rule (GLOBAL VAR)) is checked for property $x$. If it exists, then it is returned with dependencies. If the location for a particular variable is found in neither the current scope nor the global scope, using rule (UNINITIALIZED VAR) we create a new node $n_{new}$ and add it to the global scope. Similar rules apply for field accesses in rules (FIELD ACCESS) and (UNINIT FLD).

For binary operators(rule (BINARY OP)), we return the union of dependencies of both the expressions. When an object literal expression((OBJ. LIT.)) is encountered, a *summary* is computed by recursively creating heap locations for each of its properties and then creating the



$$\frac{}{(c,\sigma) \Downarrow (ns_\sigma, \bot, \emptyset, fr_\sigma, dm_\sigma, tm_\sigma)}\ \text{(CONSTANT)} \qquad \frac{}{(\texttt{this}\ ,\sigma) \Downarrow (ns_\sigma, tm_\sigma(n_\sigma), dm_\sigma(tm_\sigma(n_\sigma)), fr_\sigma, dm_\sigma, tm_\sigma)}\ \text{(THIS)}$$

$$\frac{(n_\sigma, x, n_x) \in fr_\sigma}{(x,\sigma) \Downarrow (ns_\sigma, n_x, dm_\sigma(n_x), fr_\sigma, dm_\sigma, tm_\sigma)}\ \text{(VAR)} \qquad \frac{\nexists n'_x.(n_\sigma, x, n'_x) \in fr_\sigma \quad (n_G, x, n_x) \in fr_\sigma}{(x,\sigma) \Downarrow (ns_\sigma, n_x, dm_\sigma(n_x), fr_\sigma, dm_\sigma, tm_\sigma)}\ \text{(GLOBAL VAR)}$$

$$\frac{\nexists n'_x.(n_\sigma, x, n'_x) \in fr_\sigma \quad \nexists n''_x.(n_G, x, n''_x) \in fr_\sigma \quad n_G \neq n_\sigma}{(x,\sigma) \Downarrow (ns_\sigma \cup \{n_{new}\}, n_{new}, \emptyset, fr_\sigma \cup \{(n_G, x, n_{new})\}, dm_\sigma, tm_\sigma \cup \{(n_{new}, n_G)\})}\ \text{(UNITIALIZED VAR)}\ where,\ n_{new} = fresh()$$

$$\frac{(x,\sigma) \Downarrow \sigma' \quad (n_{\sigma'}, f, n_f) \in fr_{\sigma'}}{(x.f,\sigma) \Downarrow (ns_{\sigma'}, n_f, d_{\sigma'} \cup dm_{\sigma'}(n_f), fr_{\sigma'}, dm_{\sigma'}, tm_{\sigma'})}\ \text{(FIELD ACCESS)} \qquad \frac{(x,\sigma) \Downarrow \sigma'}{(x.prot,\sigma) \Downarrow \sigma'}\ \text{(PROT ACCESS)}$$

$$\frac{(x,\sigma) \Downarrow \sigma' \quad \nexists n_f.(n_{\sigma'}, f, n_f) \in fr_{\sigma'}}{(x.f,\sigma) \Downarrow (ns_{\sigma'} \cup \{n_{new}\}, n_{new}, d_{\sigma'}, fr_{\sigma'} \cup \{(n_{\sigma'}, f, n_{new})\}, dm_{\sigma'}, tm_{\sigma'} \cup \{(n_{new}, n_{\sigma'})\})}\ \text{(UNINIT FLD)}\ where, n_{new} = fresh()$$

$$\frac{(e_1,\sigma) \Downarrow \sigma_1 \quad (e_2,\sigma) \Downarrow \sigma_2}{(e_1\, op\, e_2,\sigma) \Downarrow (ns_{\sigma_1} \cup ns_{\sigma_2}, \bot, d_{\sigma_1} \cup d_{\sigma_2}, fr_{\sigma_1} \cup fr_{\sigma_2}, dm_{\sigma_1} \cup dm_{\sigma_2}, tm_{\sigma_1} \cup tm_{\sigma_2})}\ \text{(BINARY OP)}$$

$$\frac{(e_1,\sigma) \Downarrow \sigma_1 \quad \ldots \quad (e_n,\sigma) \Downarrow \sigma_n}{(\{f_1 : e_1, \ldots, f_n : e_n\},\sigma) \Downarrow \sigma'}\ \text{(OBJ. LIT.)} \quad where, \begin{array}{ll} n_{\sigma'} = fresh() = n_{new} & d_{\sigma'} = \bigcup_{i=1}^{n} d_{\sigma_i} \\ ns_{\sigma'} = ns_\sigma \cup \{n_{new}\} \cup (\bigcup_{i=1}^{n} ns_{\sigma_i}) & fr_{\sigma'} = fr_\sigma \cup (\bigcup_{i=1}^{n} (n_{new}, f_i, n_{\sigma_i})) \\ dm_{\sigma'} = dm_\sigma \cup (\bigcup_{i=1}^{n} dm_{\sigma_i}) & tm_{\sigma'} = tm_\sigma \cup (\bigcup_{i=1}^{n} (n_{\sigma_i}, n_{new})) \end{array}$$

$$\frac{(S,\sigma'') \Downarrow \sigma'}{(\texttt{function}\ (p_1, \ldots, p_n)\{S\},\sigma) \Downarrow \sigma'}\ \text{(FUN-DEF)} \quad where, \begin{array}{l} ns_{\sigma''} = ns_\sigma \cup \{n^0_{new}\} \cup (\bigcup_{i=1}^{n} n^{p_i}_{new}) \quad n_{\sigma''} = fresh() = n^0_{new} \quad d_{\sigma''} = \emptyset \\ n^{\_\texttt{RET}}_{new} = fresh() \quad \forall i \in \{1, \ldots, n\}.n^{p_i}_{new} = fresh() \\ fr_{\sigma''} = fr_\sigma \cup \{(n^0_{new}, \_\texttt{RET}, n^{\_\texttt{RET}}_{new})\} \cup (\bigcup_{i=1}^{n} \{(n^0_{new}, i, n^{p_i}_{new})\}) \\ dm_{\sigma''} = dm_\sigma \cup (\bigcup_{i=1}^{n} \{(\_\texttt{RET}, i), (n^{p_i}_{new}, i)\}) \\ tm_{\sigma''} = tm_\sigma \cup \{(n^0_{new}, n_\sigma)\} \cup \{(n^{\_\texttt{RET}}_{new}, n^0_{new})\} \cup (\bigcup_{i=1}^{n} (n^{p_i}_{new}, n^0_{new})) \end{array}$$

$$\frac{(f,\sigma) \Downarrow \sigma'' \quad (n_{\sigma''}, \_\texttt{RET}, n') \in fr_\sigma \quad (e_1,\sigma) \Downarrow \sigma_1 \quad \ldots \quad (e_n,\sigma) \Downarrow \sigma_n}{(f(e_1, \ldots, e_n),\sigma) \Downarrow (ns_\sigma, \bot, d', fr_\sigma, dm_\sigma, tm_\sigma)}\ \text{(FUN-CALL1)} \quad where, d' = \bigcup_{i=1}^{n} (\exists (n', i) \in dm_\sigma.d_{\sigma_i})$$

$$\frac{(f,\sigma) \Downarrow \sigma'' \quad n_{\sigma''} = \bot \quad (e_1,\sigma) \Downarrow \sigma_1 \quad \ldots \quad (e_n,\sigma) \Downarrow \sigma_n}{(f(e_1, \ldots, e_n),\sigma) \Downarrow (ns_\sigma, \bot, \bigcup_{i=1}^{n} d_{\sigma_i}, fr_\sigma, dm_\sigma, tm_\sigma))}\ \text{(FUN-CALL2)}$$

Figure 3: Semantics for all core expressions except `new`.

graph where the new object node is linked to the properties with the labeled edges.

A function definition((FUN-DEF)) is treated in a similar fashion as the object literal, except that new summary locations are created for each of the function arguments and also for the return variable (i.e. $n^{\_\texttt{RET}}_{new}$). The function body is evaluated with respect to the new heap. The result of the evaluation is the new heap with the function summary attached to the node $n^{\_\texttt{RET}}_{new}$. A function call(rule (FUN-CALL1)) uses this summary to compute the node and dependencies of the return value. The return value of the function can be obtained by evaluating each of the function argument expressions, and replacing the appropriate nodes in the function summary with the values returned. If the function is not defined, then the dependencies of the return values are the union of dependencies of the individual function parameters(rule (FUN-CALL2)). A constructor expression (containing `new`) is similar to a function call, where if the object being instantiated is retrieved from the local or the global scope, then a copy of the graph starting with this object is created and returned.

**Evaluating statements:**
The statement semantics are given in Figure 4. A variable declaration(VAR. DECL.) creates a new node in the current scope. If the heap node for that variable already exists, it is replaced by this new node. The assignment statement (rules (ASSIGN1) and (ASSIGN2)) evaluates the left hand side and the right hand side expressions, replaces the node on the left hand side with the node on the right hand side. Note that conditionals in `if-then-else` and `while` statements are, of course, not evaluated as our heaps are symbolic. The while state-

$$\frac{}{(\texttt{skip}, \sigma) \Downarrow \sigma}\ (\textsc{Skip}) \qquad \frac{(C_1, \sigma) \Downarrow \sigma' \quad (C_2, \sigma') \Downarrow \sigma''}{(C_1; C_2, \sigma) \Downarrow \sigma''}\ (\textsc{Seq})$$

$$\frac{(n_\sigma, x, n_x) \in fr_\sigma}{(\texttt{var}\ x, \sigma) \Downarrow (ns, n_\sigma, d_\sigma, fr, dm_\sigma, tm)}\ (\textsc{Var.Decl.}) \qquad \begin{array}{l} where, ns = (ns_\sigma \cup \{n_{new}\}) \setminus \{n_x\} \\ fr = (fr_\sigma \setminus \{(n_\sigma, x, n_x)\}) \cup \{(n_\sigma, x, n_{new})\} \\ tm = tm_\sigma \cup \{(n_{new}, n_\sigma)\} \end{array}$$

$$\frac{(e, \sigma) \Downarrow \sigma'' \quad (x, \sigma) \Downarrow \sigma_x}{(x := e, \sigma) \Downarrow (ns, n_\sigma, d_\sigma, fr, dm, tm)}\ (\textsc{Assign1}) \qquad \begin{array}{l} where, ns = ns_{\sigma_x} \cup ns_{\sigma''} \\ fr = (fr_{\sigma''} \setminus \{(n_\sigma, x, n_{\sigma_x})\}) \cup \{(n_\sigma, x, n_{\sigma''})\} \\ dm = dm_{\sigma''} \\ tm = tm_\sigma \cup \{(n_{\sigma''}, tm_{\sigma_x}(n_{\sigma_x}))\} \end{array}$$

$$\frac{(e, \sigma) \Downarrow \sigma'' \quad (x, \sigma) \Downarrow \sigma_x \quad (x.f, \sigma) \Downarrow \sigma_f}{(x.f := e, \sigma) \Downarrow \sigma'}\ (\textsc{Assign2}) \qquad \begin{array}{l} where, n_{\sigma'} = n_\sigma \quad d_{\sigma'} = d_\sigma \\ fr_{\sigma'} = (fr_{\sigma''} \setminus \{(n_{\sigma_x}, f, n_{\sigma_f})\}) \cup \{(n_{\sigma_x}, f, n_{\sigma''})\} \\ dm_{\sigma'} = dm_{\sigma''} \cup \{(n_{\sigma''}, y) | y \in d_{\sigma_x}\} \cup \{(n_{\sigma_x}, y) | y \in d_{\sigma''}\} \\ tm_{\sigma'} = tm_\sigma \cup \{(n_{\sigma''}, n_{\sigma_x})\} \end{array}$$

$$\frac{(S_1, \sigma) \Downarrow \sigma_1 \quad (S_2, \sigma) \Downarrow \sigma_2}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma) \Downarrow (ns_{\sigma_1} \cup ns_{\sigma_2}, n_\sigma, d_\sigma, fr_{\sigma_1} \cup fr_{\sigma_2}, dm_{\sigma_1} \cup dm_{\sigma_2}, tm_{\sigma_1} \cup tm_{\sigma_2})}\ (\textsc{Cond})$$

$$\frac{(S_1, \sigma) \Downarrow \sigma'}{(\texttt{while}\ e\ \texttt{do}\ \ S_1\ \texttt{od}\ , \sigma) \Downarrow \sigma'}\ (\textsc{While1}) \qquad \frac{(S_1, \sigma) \Downarrow \sigma' \quad (\texttt{while}\ e\ \texttt{do}\ \ S_1\ \texttt{od}\ , \sigma') \Downarrow \sigma''}{(\texttt{while}\ e\ \texttt{do}\ \ S_1\ \texttt{od}\ , \sigma) \Downarrow \sigma''}\ (\textsc{While2})$$

$$\frac{(e, \sigma) \Downarrow \sigma'}{(\texttt{return}\ e, \sigma) \Downarrow (ns_{\sigma'}, n_\sigma, d_\sigma, fr_{\sigma'} \cup \{(n_\sigma, \_\_\texttt{RET}, n_{\sigma'})\}, dm_{\sigma'}, tm_{\sigma'})}\ (\textsc{Ret})$$

Figure 4: Statement semantics.

ment is interesting: we evaluate the while body till we reach a fixed point (or till we reach a fixed number of loop un-rollings) as depicted in (WHILE2). However, notice that the abstract heap is also allowed to immediately go across a while-loop (WHILE1). The semantics for the rest of the statements is standard.

Given the above rules for abstract heaps, we start analyzing the JavaScript program using an initial state consisting of a global heap, represented by node $n_G$. This global heap consists of summaries for a few built-in objects like Array. We evaluate the rules either till we converge on a least fixed-point, or till we reach a preset bound on the number of iterations.

## 4.2 Handling other features of JavaScript

**Dynamic code:** The `eval` method in JavaScript allows execution of dynamically formed code, and is widely used in browser extensions. While an accurate analysis of the structure of dynamically created code is a research topic in itself, and quite out of the scope of this paper, we cannot simply ignore `eval` statements. Our approach has been to implement a static *constant-string analysis* for strings and subject the strings that are `eval`-ed to this analysis. Our static analysis engine inserts these constant strings into the code (as though it was static code), parses it, and computes the flows for them. Strings that are not statically known but subject to `eval` are essentially ignored, and this causes our tool to be unsound (see a later note on unsoundness). In most correct extensions, an `eval`-ed statement is dynamically chosen from a set of constant-strings or taken from trusted sources. Note that if there is a flow from an *untrusted* source to an `eval`, VEX will catch this flow, as it is a vulnerable flow pattern.

**innerHTML:** Modifications of the `innerHTML` of an HTML page by the extension makes the analysis considerably more complex. For instance, if a function `a()` calls function `b()` that calls function `c()`, and `c()` makes `innerHTML` modifications, it is hard to summarize this effect in the summary of `c()`, as the source of the flow is not locally available. We handle this by creating a *symbolic* representation of the source, computing summaries of `innerHTML` using this symbolic source, and allowing outside methods to instantiate the symbolic source to a concrete source in whichever context it becomes available.

**Object properties accessed in the form of associative arrays:** In JavaScript, objects are treated as associative arrays. This means that any property of the object can be accessed using the array notation. Array indices could be constant strings, which are then evaluated to get the actual property being accessed; or they could be numbers, which indicate the property number that is being accessed; or they could be variables, that could be instantiated at run time. VEX treats these cases in a conservative manner. Whenever a property is created in the node scope, its dependencies are added to the dependencies of the node as shown in the (ASSIGN 2) rule in the Figure 4. If we cannot evaluate the array index for any reason, it would be sufficient to retrieve the dependencies of the object.

**Functions that take arbitrary number of arguments:** Some functions in JavaScript can have variable numbers of arguments. For example, the *push* method of the array can be called with any number of arguments and the arguments will be appended to the end of the array. To handle this, the summary of the *push* method has a special field indicating that it can take variable number of arguments and when the method is called, we conservatively append the dependencies of all the arguments to the dependency set of the node representing the array object.

**Browser's DOM API and XPCOM components:** These objects are treated as uninitialized variables, fields and functions. The rules (UNINITIALIZED VAR), (UNINIT FLD) and (FUN-CALL2) can be applied to their accesses. When we need to keep track of the usage of certain components we introduce the component API function arguments into the dependency set. For example the RDF datasource is accessed using the following command:

```
rdf = Components.classes
      ["@mozilla.org/rdf/rdf-service;1"]
      .getService(Components.interfaces.nsIRDFService);
```

Our analysis introduces the string *"@mozilla.org/rdf/rdf-service;1"* and the variable *nsIRDFService* into the dependency set of the left hand side variable $rdf$.

## 4.3 Unsoundness and incompleteness

A static analysis tool like VEX is inherently conservative. First, if VEX reports a flow, there may be no such feasible flow in the program (i.e. VEX can have false positives). Though VEX *over-approximates* flows and tries to perform a sound analysis, there are several aspects of the analysis which, if implemented soundly, will make the tool throw too many infeasible flows, making it useless in practice.

Consider a program where there is an `eval` of a string that is dynamically created and not determinable statically. Since this string can be assigned any value, it could be any arbitrary program that can create flows between any of the variables in scope. A sound tool must *necessarily* summarize the `eval` as causing flows from all variables to all nodes, which would generate plenty of false positives and would essentially be useless. False negatives (i.e. miss detecting programs that have a flow), are also possible because of the fact that we have several uninitialized and unsummarized objects.

VEX has several sources of unsoundness and incompleteness: handling of `eval`, handling of prototypes, handling of higher-order functions, fixed number of unrolls of loops, handling `with`-scoping, handling exceptions, etc.

# 5 Evaluation

## 5.1 VEX implementation

The VEX tool checks for two kinds of flows: one from injectable sources to executable sinks to check for script-injection vulnerabilities, and the other, also modeled as flows, that checks for unsafe programming practices. VEX is implemented in Java ($\sim$ 2000 LOC), and utilizes a JavaScript parser built using the ANTLR parser generator for the JavaScript 1.5 grammar provided by ANTLR [1]. ANTLR outputs Java-based Abstract Syntax Trees (AST) for JavaScript files, and VEX walks through the ASTs computing the flow sets from all interesting sources to all interesting sinks, in a single pass analysis, using the static analysis described in Section 4. For each sink object, VEX collects all the source objects that flow into it and checks for the occurrence of flow patterns. VEX reports these flows to the user along with the source and sink locations in the code.

**Flow patterns checked:** The current version of VEX checks for the following three flow patterns that capture flows from injectable sources to executable sinks:

- **Content Doc to Eval:** The source location is any point where the program accesses the API `window.content.document`, and the source object is the object that is returned from this call. The sink locations are `eval` statements and the sink objects are the objects being `eval`-ed.

- **Content Doc to innerHTML:** The source location and source objects for these flows are the same as above; the sink locations are the places where the extension writes directly into the DOM using `innerHTML` commands, and the sink objects are the objects being assigned by the `innerHTML` command. These DOM elements may be executable if they are in the chrome context.

- **RDF to innerHTML:** The source location and source objects are given by any retrieval of RDF objects (which are often injectable) and the sink locations

and sink objects are `innerHTML` commands as above.

Furthermore, VEX searches for the following patterns that characterize two documented unsafe programming practices that could lead to security vulnerabilities:

- **evalInSandbox object to == or !=:** This flow is meant to detect an unsafe programming practice where an object retrieved by an `eval` in a sandbox is subject to an == or != test (the recommended practice is that such objects must be tested with ===). The source location is hence any `evalInSandbox`-statement and the corresponding source objects are the objects *returned* by the `evalInSandbox` call. The sink locations are usages of == and !=, and the sink objects are the objects that are subject to these comparisons.

- **Method Call on wrappedJSObject:** Objects obtained using wrappedJSObject() commands are usually untrusted, and methods of such objects should not be called. The source locations are hence uses of wrappedJSObject() and source objects are the objects *returned* by them. Sink locations are methods calls and the sink objects are the objects whose methods are called.

The VEX tool can, of course, be adapted to other kinds of suspect flows – source and sink locations are straightforward, and the source and sink objects must be specified carefully as above.

## 5.2 Evaluation methodology

The extensions we analyzed were chosen as follows. First, in October 2008, we built a suite of extensions using a random sample of 1827 extensions from the Mozilla add-ons web site, by downloading the first extensions in alphabetical order for all subject categories. In November 2009, we downloaded 699 of the most popular extensions. The two sets had 74 extensions in common, for a total of **2452 extensions.** Our suite includes *multiple versions* of some extensions, allowing cross-version comparisons. For instance, we found a vulnerability in a new version of BEATNIK (see Section 5.4), though its authors thought the vulnerabilities in the previous version were fixed.

We extracted the JavaScript files from these extensions and ran VEX on them, using a 2.4GHz 64 bit x86 processor with a maximum heap size of 4GB for the JVM.

## 5.3 Experimental results

**Finding flows from injectible sources to executable sinks:** Figure 5 summarizes the experimental results for flows that are from injectible sources to executable sinks (the first three flows outlined above). The first column is the number of extensions that syntactically has code that could indicate such a flow, identified using a `grep`-search. For the flow "Content-doc to Eval", the `grep` was for the string 'eval('; for "Content-doc to InnerHTML" flows, the `grep` was for the string 'innerHTML'; and for "RDF to InnerHTML" flows, the search was for both the strings "'innerHTML" and "@mozilla.org/rdf/rdf-service;1". As the table shows, this search finds hundreds of suspect extensions, far more than can be examined manually.

The third column indicates the number of extensions on which VEX reports an alert with corresponding flows. On an average, VEX took only 15.5 seconds per extension.

To look for potential attacks, we manually analyzed most of the extensions with suspect flows that VEX alerted us on, spending about two hours per extension on average.

The next column reports the number of extensions on which we could engineer an attack based on the flows reported by VEX. We were able to attack six extensions, of which only three extensions were already known to be vulnerable. The attacks on Wikipedia Toolbar, Fizzle version 0.5.1 and Fizzle version 0.5.2 extensions are new, see Section 5.4 for more details.

The next column shows the extensions where the source is code from a web site, and where an attack *is possible* provided the web site can be attacked. In other words, these extensions rely on a *trusted web site* assumption (e.g., that the code on the Facebook website is safe). We think that these are valid warnings that users of an extension (and Mozilla) should be aware of; trusted web sites can after all be compromised, and the code on these sites can be changed leading to an attack on all users of such an extension.

Not all flows lead to attacks – the next set of columns describe the alerts that we were unable to convert to concrete attacks. Some flows were not exploitable as the input is *sanitized* correctly (either by the extension or the browser), preventing JavaScript injection, while others were not exploitable as the sinks do not turn out to be chrome executable contexts. These extensions are noted in the next two columns. Finally, VEX, being a conservative flow-analysis tool, does report alerts about flows that do not actually exist— there were very few of these, and are noted under the column "Non-existent flows". A discussion on flows that do not lead to attacks is given in Section 5.5.

As noted in the last column, there were 13 extensions with VEX alerts that were too complex(or obscurely written) for us to manually analyze for an attack; we do not know whether attacks on these are possible or not.



| Flow Pattern | `grep` Alerts | VEX Alerts | Attackable Extensions | Source is trusted website | Not Attackable | | | Unanalyzed |
|---|---|---|---|---|---|---|---|---|
| | | | | | *Sanitized input* | *Non-chrome sinks* | *Non-existent flows* | |
| Content Doc to `eval` | 430 | 13 | 2* | 1 | 0 | 3 | 5 | 2 |
| Content Doc to `innerHTML` | 534 | 46 | 0 | 14 | 6 | 6 | 9 | 11 |
| RDF to `innerHTML` | 60 | 4 | 4** | 0 | 0 | 0 | 0 | 0 |

Attackable Extensions: * WIKIPEDIA TOOLBAR V-0.5.7, WIKIPEDIA TOOLBAR V-0.5.9 ,
** FIZZLE V-0.5, FIZZLE V-0.5.1, FIZZLE V-0.5.2 & BEATNIK V-1.2

Figure 5: Flows from injectible sources to executable sinks.

| Unsafe Programming Practices | `grep` Alerts | VEX Alerts |
|---|---|---|
| `evalInSandbox` Object to == or != | 107 | 3 |
| Method Call on `wrappedJSObject` | 269 | 144 |

Figure 6: Results for unsafe programming practices.

**Finding unsafe programming practices:**
The results of the second set of experiments for flows that characterize the two unsafe programming practices of checking equality on objects evaluated in a sandbox and calling methods of unwrapped JS objects are shown in Figure 6.

The first column denotes the flow-pattern, the second column shows the number of extensions that had a `grep` pattern for the strings 'evalInSandbox' and 'wrappedJSObject', respectively. The third column shows the number of extensions that VEX alerts. Note that these flows correspond to unsafe programming practices declared by Mozilla for extension writers, and hence should be avoided. We analyzed 15 of the alerts and found that all of the flows we inspected were feasible and real, but we were unable to manually confirm the remainder because there were too many alerts to examine.

## 5.4 Successful attacks

**Attack scripts:** All our attack scenarios involve a user who has installed a vulnerable extension who visits a malicious page, and either automatically or through invoking the extension, triggers script written on the malicious page to execute in the chrome context. Figure 7 illustrates an attack payload that can be used in such attacks: this script displays the folders and files in the root directory. The attack payloads could be much more dangerous, where the attacker could gain complete control of the affected computer using XPCOM API functions. More examples of such payloads are enumerated in the white-paper given in [13].

Let us now explain the various attacks we found on web extensions:

**Wikipedia Toolbar, up to version 0.5.9**
If a user visits a web page with the directory display attack script in its <head> tag, and clicks on one of the Wikipedia toolbar buttons (unwatch, purge, etc.), the script executes in the chrome context. The attack works because the extension has the code given in Figure 8 in its toolbar.js file.

```
<script>
var root = Components.classes
["@mozilla.org/file/local;1"].createInstance
(Components.interfaces.nsILocalFile);
try {
root.initWithPath("/."); // for Linux or Mac
}catch (er){
root.initWithPath("\\\\."); // for Windows
}
var drivesEnum = root.directoryEntries,
drives = [];
while (drivesEnum.hasMoreElements()) {
drives.push(drivesEnum.getNext().
QueryInterface(Components.interfaces.
nsILocalFile).path);
}
alert (drives);
</script>
```

Figure 7: Attack script to display directories.

```
script = window._content.document.
getElementsByTagName(``script")[0].innerHTML;
eval (script);
```

Figure 8: Wikipedia toolbar code.

The first line gets the first <script> element from the web page and executes it using `eval`. The extension developer assumes the user only clicks the buttons when a Wikipedia page is open, in which case <script> may not be malicious. But the user might be fooled by a malicious Wikipedia spoof page, or accidentally press the button on some other page, VEX led us to this previously unknown attack, which we reported to the devel-

opers, who acknowledged it, patched it, and released a new version. This resulted in a new **CVE vulnerability (CVE-2009-41-27).** The fix involved inserting a conditional in the program to check if the url of the page is on the wikipedia domain and evaluating the script only if this is true.

**bookmarks.js:**

```
1. function Bookmarks(){
2.  var  bookmarks = new  Array();
3.  this.load = function(){
4.   bookmarks = new  Array();
5.   var  rdf = Components.classes[
      "@mozilla.org/rdf/rdf-service;1"]
     .getService(Components.interfaces.nsIRDFService);
6.   var  bmds = rdf.GetDataSource("rdf:bookmarks");
7.   var  iter = bmds.GetAllResources();
8.   while (iter.hasMoreElements()){
9.    var  element = iter.getNext();
10.  bookmarks.push(
       {name:element.name, url:element.url});
11. } } }
```

**sys.js:**

```
12.    var  sys = new  Sys();
13.    function Sys() {
14.     var  bookmarks = null;
15.     this.startup = function() {
16.      bookmarks = new  Bookmarks();
17.      bookmarks.load();
18.      ui.buildFeedList(); }
19.    this.getBookmarks(){
20.     return  bookmarks; } }
```

**ui.js:**

```
21.   var  ui = new  Ui();
22.   function Ui() {
23.    this.buildFeedList = function() {
24.     var  bm = sys.getBookmarks();
25.     for (var  i=0; i<bm.size(); i++) {
26.      var  mark = bm.get(i);
27.      html += <p> mark.name; }
28.     div.innerHTML = html; } }
```

Figure 9: FIZZLE vulnerability code.

**Fizzle versions 0.5, 0.5.1, 0.5.2**

FIZZLE is a RSS/Atom feed reader that uses Livemark bookmark feeds. Vulnerability report CVE-2007-1678 explains that FIZZLE VER.0.5 allows remote attackers to inject arbitrary web scripts or HTML via RSS feeds. FIZZLE's RSS feeds are obtained from the bookmarks' RDF resource, using the XPCOM RDF service. The author of FIZZLE purportedly *fixed* this vulnerability in the next version; however, VEX signaled the presence of a flow, and we found that the sanitization routine that the programmer wrote was flawed, and the extension can be attacked using suitably encoded scripts. These new attacks for FIZZLE VER 0.5.1 and FIZZLE VER 0.5.2 were not known before, to the best of our knowledge.

Figure 9 gives a highly simplified version of FIZZLE, to show its information flows. When the user clicks on the FIZZLE extension toolbar to see the feeds, FIZZLE is initialized, i.e., `sys.startup()` on line 15 is called. This method loads the bookmarks from the Firefox bookmarks folder. The title and URL of the feeds are obtained from the bookmarks' RDF resource and then stored in an array in FIZZLE when `bookmarks.load()` is called. After the bookmarks are loaded, `ui.buildFeedList()` is called. In this method, the bookmark array is accessed on line 24 and the elements are added to a variable named `html` on line 27. This `html` variable is then assigned to the `innerHTML` property of the $\langle div \rangle$ tag of an HTML page. This page is then displayed in a frame in the browser. The attack happens when a malicious RDF file is loaded, where the `name` element of the feed contains JavaScript. Assigning a specially crafted script to the `innerHTML` property at line 28 results in the script being executed under chrome privileges.

To detect this kind of attack, we must be able to determine that the information that flows into the `html` variable and eventually into the `innerHTML` property is from the bookmarks' RDF resource. It is difficult to detect this manually, because most extensions are encoded in many separate JavaScript files spread across multiple directories, and the routines defined in these files have complex interactions with each other. Even the example shown in Figure 9 is spread over three different JavaScript files, and we have omitted many lines of code from the functions shown. As mentioned earlier, VEX users can define summaries for library functions, or just rely on default summaries. Given a function summary for the `push` method of the `Array` object defined in the XPCOM library, VEX detects that FIZZLE has flows from the RDF service to `innerHTML`.

**Beatnik version 1.2**

BEATNIK is another RSS reader with the same kind of problematic flow as FIZZLE, documented in CVE-2007-3110 for BEATNIK version 1.0. In the Mozilla add-ons page for the subsequent version of BEATNIK, the extension developer said he had sanitized the RSS feed input. VEX found that there were still flows from the bookmarks' RDF to the `innerHTML` property in BEATNIK version 1.2, because VEX currently does not consider declassification via sanitization. Our manual examination showed the new sanitization to be inadequate. The sanitization parses the feed input and checks whether the nodes contain script. If the feed contains only text nodes, it is appended to the RSS feed title; otherwise it is discarded. By encoding the $\langle$ and $\rangle$ tags as their HTML entity names, we can fool this routine. If we name the RSS feed as follows:

```
Title &lt; /a &gt&lt; img  src =
&quot;&quot; onerror= 'CODE  FROM  FIGURE  7'
& gt; Beatnik &lt;/img&gt; &lt; a &gt;
```

the string is converted into

```
Title </a> < img src =" " onerror= 'CODE
FROM FIGURE 7'> Beatnik </img> <a>
```

and results in an attack. To the best of our knowledge, this attack has not been reported thus far. One must understand the extension code to form these attack strings; in this case, the <a> tag had to be closed at the beginning of the string and opened again at the end for the script to work.

### 5.5 Flows that do not result in attacks

Figure 10 gives several examples of the suspect flows that we manually analyzed and for which either trusted sources were assumed by the extension or we could not find attacks.

The first set has extensions reading values from websites or sources it trusts, and the values flow to `eval`, `innerHTML`, or `evalInSandbox`. Of course, if the trusted sources are compromised, then the extensions may become vulnerable.

The second set illustrates examples where the input was sanitized between the source and the sink (we do not know for sure that the sanitization is adequate, but we were unable to attack it). The third set of extensions had non-chrome sinks. The last two examples show false positives where the flows reported by VEX do not exist. These false alarms are because of the way VEX handles variable dependencies imprecisely. For example, the last alarm is caused by the rule ASSIGN2 in Figures 3 and 4, which conservatively adds the dependencies of variable $x$ to field $f$.

## 6 Related work

Maffeis *et. al.* [27] proposed a small-step operational semantics for JavaScript, using which they analyze security properties of web applications. This operational semantics is then useful for generating safe subsets of JavaScript and to manually prove that the so-called safe subsets of JavaScript are in fact vulnerable to certain attacks [28]. Our operational semantics is inspired by their approach, although we take an alternate approach of abstracting the primitive values in the program. This helps us in proposing a precise information flow analysis approach for a non-trivial JavaScript program. More recently, Guha *et. al.* [18] also provide an operational semantics for JavaScript (albeit without semantics for *eval*) with the goal of making it easier to prove properties about the JavaScript programs.

Recent work by Ter Louw et al. [25] highlights some of the potential security risks posed by browser extensions, and proposes run time support for restricting the interactions between browsers and extensions. Our techniques are complementary to these techniques since, as our experiments show, even restricted interfaces can still be susceptible to security vulnerabilities.

Most recent work on the security of browser extensibility mechanisms focuses on plugin security. Plugins are external applications hosted within the browser that are used to render non-HTML content, such as Flash videos. The first work to examine security issues for browser plugins was Janus [14], which discusses sandboxing techniques for browser-helper applications, such as PDF viewers. More recently, the OP [15] and Gazelle [16] web browsers tackle this same issue by applying many of the principles from the original Janus work to modern browser plugins.

The general idea of secure extensibility has been studied by the systems community with projects that focus on providing secure extensions for operating systems via type safe programming languages [5, 31, 36], proof-carrying code [29], new OS abstractions [10], and software fault isolation [11]. To date, none of these techniques have been adapted to address the special security needs of web browser extensibility mechanisms.

*Static information flow analysis* has been used in a number of previous projects. The work proposed in [2] tracks whether various variables in the program are independent from each other both through explicit and implicit flows. Researchers have employed static analysis for *web applications* with the goal of identifying and preventing cross-site scripting attacks [26]. For example, Pixy [21] is a taint based static analyzer for PHP that detects flows; WebSSARI [19] offers similar facilities. Vogt et al. [32] propose combining static and dynamic techniques to prevent cross-site scripting. Xie and Aiken propose a static analysis of PHP for SQL injection vulnerabilities [34]. Livshits and Lam develop flow-insensitive static analysis tools for security properties [24].

More recently, researchers have developed a *flow-insensitive* static information flow methods for JavaScript [7, 17]. In contrast, VEX's analysis is *flow-sensitive* and *context-sensitive*. In [7] the authors essentially perform a *flow-insensitive* static analysis on the code, and delegate analysis of dynamic code to runtime checks. Furthermore, their analysis is *context-*

| Classification | Extension | Flow pattern/Unsafe practice | Explanation |
|---|---|---|---|
| Source is trusted website | TWITZER_-_TWITTER_MORE! v.1.3 | Content Doc to `innerHTML` | Works only when on Twitter |
| | ANSWERS v-2.3.50 | Content Doc to `innerHTML` | Works only on answers.com |
| | MYSPACE_FRIEND_RENAMER v-.86 | Content Doc to `innerHTML` | Fetches friend names from prefs.js, where they are stored during instantiation |
| Sanitized Input | GIRL_IN_WONDERLAND v-0.808 | Content Doc to `innerHTML` | Assigns a Flash URL to `innerHTML` of an element on the page, and sanitizes the URL before assignment; is the sanitization complete? |
| | AUTOSLIDESHOW v-0.3.4 | Content Doc to `innerHTML` | Has a flow from the image name urls to `innerHTML`. The extension did not sanitize the inputs in any way. However, the Firefox DOM API methods encoded the urls when they were being handled by the extension. |
| Non-chrome sinks | UNHIDE_FIELDS v-0.2e | Content Doc to `innerHTML` | Creates a frame on top of the current content document and displays the hidden fields in a page in that frame |
| | WEB_DEVELOPER v-1.1.6 | Content Doc to `innerHTML` | Generates a non-chrome document in a new tab or window and appends the stylesheet information of a page as a node in this page |
| Non-existent flows | POWER_TWITTER v-1.37 | Content Doc to `eval` | Has document, content and window dependencies, but they are chrome elements, not content |
| | INTERCLUE v-1.5.7 | Content Doc to `eval` | Caused by the ASSIGN1 rule |

Figure 10: Example extensions.

*insensitive*, which could generate a lot of false-positives if used for analyzing browser extensions. VEX does not delegate any work to runtime checks. Guarnieri *et. al.* [17] popose a mostly-static enforcement for JavaScript analysis. Their threat model is that of a malicious JavaScipt widget that could run in the same page as a hosting site and which may contain code obfuscation. Their policies are based on searching for forbidden objects or methods in the code which requires an accurate pointer analysis which they define.

Several *dynamic* analysis techniques with static instrumentation have been proposed for JavaScript to check information-flow properties [35, 22]. *SABRE* [9] is a framework for dynamically tracking in-browser information flows for analyzing JavaScript-based browser extensions. We believe that dynamic techniques are not the best choice for vetting web extensions, as we think it is best to analyze extensions statically before they are unleashed on ordinary users. However, dynamic techniques that *prevent* certain script injection attacks can be useful when enforced by the web browser. The drawback is that the web browser must choose an appropriate action to take when it detects a questionable flow. Querying the user may not be wise, and default options may become too restrictive. Additionally, SABRE imposes a performance and memory overhead to the browser because of the need to keep track of the security label for every JavaScript object inside the browser.

Recently, Freeman and Liverani from Security Assessment have written a white paper [12] detailing the possible attacks on Firefox extensions. We are currently in the process of extending VEX to incorporate some of the source/sink pairs shown in that paper.

## 7 Conclusions

Our main thesis is that most vulnerabilities in web extensions can be characterized as explicit flows, which in turn can be statically analyzed. VEX is a proof-of-concept tool for detecting potential security vulnerabilities in browser extensions using static analysis for explicit flows. VEX helps automate the difficult manual process of analyzing browser extensions by identifying and reasoning about subtle and potentially malicious flows. Experiments on thousands of extensions indicate that VEX is successful at identifying flows that indicate potential vulnerabilities. Using VEX, we identify three previously unknown security vulnerabilities and three previously known vulnerabilities, together with a variety of instances of unsafe programming practices.

The most important future direction we envision is to extend the VEX analysis in three ways. First, the static analysis can benefit from a points-to analysis that is more precise on certain aspects of JavaScript such as higher-order functions, prototypes, and scoping. The second important extension is to define a more complete set of flow-patterns (sources and sinks) that capture vulnerabilities. In preliminary work, we have found 16 more known vulnerabilities, of which 14 can be characterized using information flow-patterns. Identifying statically these source-sink pairs and adding them to VEX would yield a more comprehensive tool. In the direction of reducing false positives, automatically building attack vectors for statically discovered flows can help synthesize attacks; a key challenge in achieving this would be in handling sanitization routines effectively [3, 30].



## 8 Acknowledgments

We thank Chris Grier and Mike Perry who directed us to the Firefox extension vulnerabilities. We thank Wyatt Pittman and Nandit Tiku for gathering and analyzing the known Firefox extension vulnerabilities. We thank all the reviewers of this paper for their helpful comments and suggestions. This research was funded in part by NSF CAREER award #0747041, NSF grant CNS #0917229, NSF grant CNS #0831212, grant N0014-09-1-0743 from the Office of Naval Research, and AFOSR MURI grant FA9550-09-01-0539.

## References

[1] ANTLR Parser Generator. `http://www.antlr.org`.

[2] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In R. Giacobazzi, editor, *SAS 2004*, volume 3148 of *LNCS*, pages 100–115. Springer-Verlag, 2004.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2010.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 267–283, December 1995.

[6] A. Boodman. The Greasemonkey Firefox extension. `https://addons.mozilla.org/en-US/firefox/addon/748`.

[7] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In M. Hind and A. Diwan, editors, *PLDI*, pages 50–62. ACM, 2009.

[8] CrYpTiC MauleR. Fizzle RSS Feed HTML Injection Vulnerability. `http://www.securityfocus.com/bin/23144`.

[9] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *ACSAC'09: Proceedings of the 25th Annual Computer Security Applications Conference*, pages 382–391, December 2009.

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM.

[11] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, pages 75–88. USENIX Association, 2006.

[12] N. Freeman and R. S. Liverani. Cross context scripting with Firefox, April 2010. `http://www.security-assessment.com/files/whitepapers/Cross_Context_Scripting_with_Firefox.pdf`.

[13] N. Freeman and R. S. Liverani. Exploiting cross context scripting vulnerabilities in Firefox, April 2010. `http://www.security-assessment.com/files/whitepapers/Exploiting_Cross_Context_Scripting_vulnerabilities_in_Firefox.pdf`.

[14] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 1996 USENIX Security Symposium*, pages 1–13, July 1996.

[15] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[16] C. Grier, H. J. Wang, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 2009 Usenix Security Symposium*, 2009.

[17] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of USENIX Security '09*, pages 151–168, 2009.

[18] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP*, Lecture Notes in Computer Science. Springer, 2010.

[19] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, pages 40–52, New York, NY, USA, 2004. ACM.

[20] IAOSS. NoScript Firefox extension. `http://noscript.net/`.

[21] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceesings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[22] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. JavaScript instrumentation in practice. In *APLAS '08*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] R. S. Liverani and N. Freeman. Abusing Firefox extensions, Defcon 17, July 2009.

[24] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[25] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Extensible web browser security. In B. M. Hämmerli and R. Sommer, editors, *DIMVA*, volume 4579 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2007.

[26] G. A. D. Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana. Identifying cross site scripting vulnerabilities in web applications. In *WSE '04*, pages 71–80, Washington, DC, USA, 2004. IEEE Computer Society.

[27] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In G. Ramalingam, editor, *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2008.

[28] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[29] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[30] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy*, 2010.

[31] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI*, pages 213–227, 1996.

[32] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*. The Internet Society, 2007.

[33] C. Waterson. RDF in fifty words or less. `https://developer.mozilla.org/en/RDF_in_Fifty_Words_or_Less`.

[34] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[35] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 237–249. ACM, 2007.

[36] F. Zhou, J. Condit, Z. R. Anderson, I. Bagrak, R. Ennals, M. Harren, G. C. Necula, and E. A. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 45–60. USENIX Association, 2006.



# Securing Script-Based Extensibility in Web Browsers

*Vladan Djeric, Ashvin Goel*
*University of Toronto*

## Abstract

Web browsers are increasingly designed to be extensible to keep up with the Web's rapid pace of change. This extensibility is typically implemented using script-based extensions. Script extensions have access to sensitive browser APIs and content from untrusted web pages. Unfortunately, this powerful combination creates the threat of privilege escalation attacks that grant web page scripts the full privileges of extensions and control over the entire browser process.

This paper makes two contributions. First, it describes the pitfalls of script-based extensibility based on our study of the Firefox web browser. We find that script-based extensions can lead to arbitrary code injection and execution control, the same types of vulnerabilities found in unsafe code. Second, we propose a taint-based system to track the spread of untrusted data in the browser and to detect the characteristic signatures of privilege escalation attacks. We evaluate this approach by using exploits from the Firefox bug database and show that our system detects the vast majority of attacks with almost no false alarms.

## 1 Introduction

Most web browsers today provide powerful extensibility features, including native and script-based extensions. Native extensions (or plugins) are typically used when performance is critical (e.g., virtual machines for Java, Flash, media players, etc.), while script extensions ensure memory safety and have the advantage of being inherently cross-platform and amenable to rapid development. Examples of popular script extensions include the Firefox Adblock extension [1] that filters content from blacklisted advertising URLs, and Greasemonkey [4] that allows users to install arbitrary scripts in web pages for customization or to create client-side mashup pages.

Script extensions must have access to both sensitive browser APIs and content from untrusted web pages. For example, Adblock must be able to access the local disk to store its URL blacklist and access web pages to filter their content. This combination is needed for writing powerful extensions, but it creates challenges for securely executing web page scripts. Specifically, when extensions interact with web pages, there is a risk of a privilege escalation attack that grants web page scripts the full privileges of script extensions and control over the entire browser process. Privilege escalation vulnerabilities are perhaps even *more* critical than memory safety vulnerabilities because script-based attacks can often be executed reliably.

Our goals in this paper are two-fold: 1) understanding the nature of script-based privilege escalation vulnerabilities, 2) proposing methods to secure the Firefox browser against them. Privilege escalation vulnerabilities are common in Firefox, and comprise roughly a third of the critical vulnerability advisories. They arise from unsafe extension behaviors or bugs in the Firefox security mechanisms that regulate interactions between trusted native or extension scripts and untrusted web page scripts. These vulnerabilities have appeared regularly in every version of the browser and exist even in the latest versions. This is despite continuing effort from a dedicated team of security developers that have progressively improved the browser security model.

The Firefox security model consists of a combination of stack inspection and one-way namespace isolation. The stack inspection mechanism, implemented at the boundary of the script and native code, regulates accesses to sensitive native interfaces based on the principals of the caller. For example, a local file access is denied if the current stack contains a frame associated with an untrusted principal.[1] Namespace isolation is used to enforce the same-origin policy for web page scripts. This policy limits interactions between scripts and documents loaded from different origins. The namespace isolation is *one way* in that script extensions

[1] A principal represents the code's origin and, for web page scripts, it consists of a scheme, host, port combination.

are privileged and allowed to access content namespaces, but web page scripts should not be able to obtain a reference to the privileged namespace. This policy is designed to enforce the same-origin policy and defend against privilege escalation attacks.

These security mechanisms are well understood, but they have two flaws: 1) relying entirely on principals as a measure of trustworthiness for stack inspection, and 2) depending on one-way namespace isolation to work correctly. In practice, an exploit can leverage browser bugs or vulnerable extensions to confuse the browser into assigning wrong principals to code or executing data or code with wrong principals, thus defeating stack inspection. Second, reference leaks can occur because of interactions between privileged and unprivileged scripts, compromising namespace isolation and allowing unprivileged scripts to affect the execution of privileged scripts. As a result, we find that arbitrary code injection and execution control vulnerabilities that commonly exist in unsafe code can also occur with script-based extensibility.

Based on the flaws described above, our solution for securing the Firefox browser consists of combining tainting with the existing stack-based security model. Our approach guarantees that tainted data will not be executed as privileged code. Tainting *all* data from untrusted origins and propagating the tainted data throughout the browser provides a much stronger basis for making security decisions. In essence, our attack detectors "second guess" the security decisions of the browser by taking into account one additional piece of information, i.e. the taint status. This solution is conceptually simple and well-suited for the browser's security model because namespace isolation already provides a security barrier between the taint sources in content namespaces and privileged code residing in extension namespaces. As a result, we show that it is unlikely that attacks will be detected erroneously, even if we fully taint all data and scripts from web pages.

The contributions of this paper are two-fold: 1) we analyze and classify script-based privilege escalation vulnerabilities in the commonly used Firefox browser, 2) we use taint-based stack inspection to design effective signatures for script-based exploits and evaluate this approach. We use Firefox version 1.0 for the evaluation because it has several privilege escalation vulnerabilities and easily-available exploits. Our results show that we can detect the vast majority of attacks with almost no false alarms and modest overhead.

Below, Section 2 provides background on the Firefox security model. Section 3 presents our classification of privilege escalation vulnerabilities and sample exploits. Section 4 describes our taint-based approach for securing script-based extensibility. Section 5 provides an evaluation of our approach. Section 6 describes related work in the area and Section 7 presents our conclusions and describes future work.

## 2 The Firefox Browser

In this section, we provide an overview of the Firefox architecture and its security model.

### 2.1 Architecture

Figure 1 shows a simplified version of the Firefox architecture relevant to this work. The basic browser functionality is provided by native C++ components written using Mozilla's cross-platform component model (XPCOM). XPCOM components implement functionality such as file and socket access, the document object model (DOM) for representing HTML documents, and higher-level abstractions, such as bookmarks, and expose this functionality via the XPIDL interface layer. The Script Security Manager (SSM) is an XPCOM component responsible for implementing the browser's security mechanisms.

The JavaScript interpreter accesses XPCOM functionality via the XPConnect translation layer. This layer allows the interpreter and the XPCOM classes to work with each others data types transparently. XPConnect also serves as the primary security barrier for enforcing the browser's same origin policy and restricting access to sensitive XPCOM interfaces.

Firefox's script extensions and privileged UI scripts, shown in Figure 1, are loaded from local files through URIs with the "chrome" protocol. They are privileged and have access to a greater number of XPCOM interface methods than web page scripts and are not subject to the browser's same origin policy. Similar to other browsers, Firefox also supports native plugins for Java, Flash, etc.



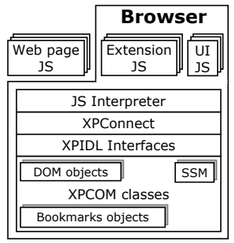



Figure 1: The Firefox architecture.

Although potential security vulnerabilities can exist within plugin implementations, we do not address them. However, with appropriate sandboxing of plugins [14, 23], we would be able to monitor any script interactions with the plugins.

### 2.2 Security Model

Firefox primarily uses two security schemes, namespace isolation and a subject-verb-object model based on stack inspection. Namespace isolation is used to enforce the same origin policy for web page scripts, and stack inspection regulates access to sensitive XPCOM components. We describe each in more detail below.

#### 2.2.1 Namespace Isolation

The browser runs scripts within an object namespace that defines the objects available to the script. A window object lies at the root of the namespace for each web page. For example, web page scripts manipulate HTML by invoking the DOM methods of the document object that is a property of this window object.

The browser enforces the same origin policy by running web page scripts from different web pages in different namespaces. These scripts are only allowed to access other namespaces from the same origin (described below). Extension scripts are allowed to access all content namespaces. Extension namespaces are hidden from the web page scripts, and extensions are not expected to invoke web page scripts directly.

#### 2.2.2 Subject-Verb-Object Model

Firefox uses a "Subject-Verb-Object" access control model. The subject is the principal of the currently executing code, the verb is one of a limited number of operations (e.g., call a function F, get a property A, set a property B), and the object is the principal of the object that is the target of the operation. This security mechanism is implemented in the Script Security Manager, and invoked by XPConnect to regulate access to sensitive XPCOM interfaces and by the interpreter to limit access to sensitive functions and object properties.

The principal of a web page script is defined by the origin of the document containing the script (its protocol, hostname, and port). The Script Security Manager determines the subject principal by walking down the JavaScript stack until it finds a stack frame with a script principal. The object principal is determined by walking up the object's parent chain (scope chain) in its namespace until an ancestor object with a principal is found. For web pages, the object's parent chain leads to a top-level HTML document associated with the window object.

## 3 Script-Based Privilege Escalation

Privilege escalation vulnerabilities are created by unsafe extension behaviors or bugs in the Firefox security mechanisms that regulate interactions between privileged and unprivileged code. In this section, we first discuss different classes of script-based privilege escalation vulnerabilities and then describe examples of real vulnerabilities.

### 3.1 Vulnerability Classification

Our analysis of the Firefox bug database revealed four main classes of privilege escalation vulnerabilities: code compilation, luring, reference leaks and insufficient argument sanitization. Most of the known Firefox vulnerabilities can be attributed to one or more of these classes.

#### 3.1.1 Code Compilation Vulnerabilities

Similar to cross-site scripting (XSS) vulnerabilities that occur in web sites, code compilation vulnerabilities allow arbitrary strings from content namespaces to be compiled into JavaScript bytecode with privileged principals. Unlike a statically typed language such as Java, JavaScript allows arbitrary strings to be converted into byte code at runtime through `eval` and eval-like functions such as `setTimeout`. The `eval` function compiles a string into byte code and executes it with

the principal of the calling script, even if the string was obtained from a different namespace. Code compilation vulnerabilities occur if attackers can trick privileged code into compiling strings supplied by the attacker, or if they can find bugs in the rules for assigning principals to newly compiled byte code. For example, it can be dangerous for privileged code to load URIs from untrusted namespaces as the URIs are capable of carrying script code inline. For example, the "javascript" protocol (e.g., `javascript:alert('Hello World');`) allows executing text after the protocol name as a script in the current namespace.

This problem may seem simple, but it has been the cause of several security bugs in Firefox. For example, even after vulnerable code was patched to sanitize URIs before loading them, exploits were possible because they did not account for nested URIs such as `view-source:javascript:`.

#### 3.1.2 Luring Vulnerabilities

Luring vulnerabilities allow malicious scripts to trick privileged code into calling a privileged function of the attacker's choosing instead of the intended callee. Stack inspection prevents unprivileged scripts from calling the privileged functions directly, so malicious scripts must lure privileged code into making these calls.Luring is possible because script extensions routinely access DOM objects in content namespaces. These DOM objects are simply JavaScript wrappers for native XPCOM objects with well-defined, native interfaces. However, JavaScript's flexibility allows web page scripts to modify these wrapper objects. In versions of Firefox after 1.0.3, privileged code is protected by automatically created "safety wrappers" that hide any wrapper changes made by untrusted code. However, if the safety wrapper code contains bugs (as has often been the case), privileged code again becomes vulnerable to luring attacks.

In order to execute privileged code, an attacker can choose one of three possible kinds of callees: 1) an eval-like native function, 2) a privileged function accidentally leaked into the content sandbox (see next section), or 3) a privileged native method that legitimately exists in content namespaces. The third category consists of XPCOM methods that are visible to ordinary web page scripts because they are meant to be invoked by digitally signed web page scripts. For example, the `preference()` method of the `navigator` object allows privileged scripts to read or write the browser's configuration, such as the browser's homepage and security settings. Ordinary web page scripts cannot invoke the sensitive `preference()` method directly, but since every function is also an object in JavaScript, web page scripts can obtain an object reference to this method and potentially trick buggy privileged code into invoking the reference.

#### 3.1.3 Reference Leak Vulnerabilities

Reference leak vulnerabilities occur when web page scripts gain access to references in the extension namespace [11]. These leaks are compromises in the isolation between privileged and unprivileged namespaces. They allow an attacker to modify data or code defined in a privileged namespace and call arbitrary functions within the privileged namespace, potentially leading to arbitrary execution control. Reference leaks are dangerous because privileged code that depends on namespace isolation may become accessible to web page scripts or it may become vulnerable to code compilation or luring attacks. Reference leaks can occur due to bugs in native code that deals with namespaces. Also, careless extensions may place references to privileged objects in an untrusted namespace. Finally, reference leaks can lead to cross-principal confidentiality violations, but we do not address confidentiality in this paper.

#### 3.1.4 Insufficient Argument Sanitization

Vulnerabilities can also occur if a browser extension uses unsanitized data from untrusted namespaces as arguments to privileged XPCOM APIs. For example, if an extension used to download Flash videos from web pages uses the name of the movie file on a web page as part of the local filename to which the file is saved, it may be open to directory traversal attacks (e.g., using "../" to access normally inaccessible directories) that would not be detected by the browser's stack inspection mechanism. If the overwritten file were an extension JavaScript file, it would lead to a privilege escalation attack. This specific class of vulnerability has not been documented in the Firefox bug database, but we consider it a likely vulnerability for extensions.



```
onLinkIconAvailable: function(Href)
{
  if (favIcon && ...)  {
    favIcon.setAttribute("src",
      Href);
  }
}
```

Figure 2: Target code invoked when a LINK tag is found in the current web page.

### 3.2 Examples

We describe some examples of privilege escalation vulnerabilities from the Firefox bug database to show that these vulnerabilities can be subtle and easy to overlook.

#### 3.2.1 URI Code Injection

Figure 2 shows an example of browser JavaScript containing a code compilation vulnerability that can lead to URI code injection (Bug 290036). This GUI code displays a favicon (16x16 pixel icon) image next to the browser's URL bar. Normally the icon's URI, which is specified by the current web page, would be the HTTP address of the favicon image, but a malicious web page can specify a "javascript" protocol URI. When the privileged UI code attempts to load the image by setting the `src` property of the icon container to the `Href` URI, it will inadvertently execute script code. This code will be compiled with the unprivileged principals of the URI, but it will have access to the privileged UI namespace, allowing reference leaks, which can then be used for other attacks (e.g., see Section 3.2.4). This vulnerability occurs because the native code implementing the icon container and the compilation function are unaware of the origin of the `Href` argument.

#### 3.2.2 Compilation with Wrong Principals

Figure 3 shows code that exploits a code compilation and a reference leak vulnerability to create a dynamically-defined function (`clonedFunction`) with elevated privileges. The `eval` function compiles and executes the `evalCode` string with the unprivileged principal of the web page. However, the attacker has also supplied a second argument that specifies the namespace for name resolution during the string evaluation. Normally, this argument does not cause a problem because it belongs to the same namespace as the caller's namespace. However, `xbl_object` is a benign reference leak from a privileged namespace.

```
evalCode = "clonedFunction = \
    function deliverPayload(){...}; \
    clonedFunction()";
myElem = document.getElementById
                    ("myMarquee");
xbl_object = myElem.init.call;
eval(evalCode, xbl_object);
```

Figure 3: Exploit code that allows untrusted functions to be associated with privileged principals.

Exposing `xbl_object` is a reference leak, but it is not sufficient for an attack because the interpreter invokes `eval` with the correct caller's principals. However within `eval`, once run, the `evalCode` byte code gets access to a privileged namespace. This access by itself is still not a problem because `evalCode` runs with the web page principals, and thus will not be able to get past the stack inspection checks. Similarly, invoking `deliverPayload` directly within `evalCode` would not be problematic.

The exploit occurs when `evalCode` creates a function referenced by `clonedFunction`. The interpreter creates a new function object in the privileged namespace that is a clone of `deliverPayload`. When a function is created by cloning, its principal is set to its object principal, as described in Section 2.2.2. When the cloned function is invoked, it executes its payload with elevated privileges. In effect, this exploit attaches a user-supplied function to a privileged namespace, making it appear privileged to the security manager. This vulnerability occurs because the implementation of `eval` did not check that it was compiling code from one principal and executing it within the namespace of a more privileged principal.

The patch for this vulnerability added a check to `eval` to ensure that the principal of the caller subsumes the object principal of the second argument. However, it was discovered that this patch could be bypassed by invoking `eval` indirectly using the timer method `setTimeout`. When the natively-

```
var code = "...  payload ...";
document.body.__defineGetter__
  ("localName", Script(code));
```

Figure 4: Simplified exploit code for Bug 289074.

implemented timer fires, there are no JavaScript frames left on the stack, so the caller's principal is the fully privileged principal of the native timer code. The next patch prevented `eval` from being called directly by native code. Further patches were needed to fix other attacks on `eval`.

#### 3.2.3 Luring Privileged Code

Figure 4 shows the exploit code for a luring attack. This exploit would trigger if the `document.body.localName` property is read by the UI code. This code tricks the privileged code into working with a different property than the one it expects by associating a getter function with a native DOM object property (`localName`). Furthermore, the `Script` object behaves like an eval-like function that allows strings to be precompiled and executed with the privileges of the caller's principal.[2] The consequences are equivalent to privileged JavaScript executing a string of the attacker's choosing, although no code is compiled in the privileged namespace. This vulnerability occurs because the caller accesses an overridden property.

This problem was so widespread in Firefox 1.0 that it motivated developers to implement the "safety wrapper" mechanism that allows privileged scripts to work with native DOM objects without being exposed to any modifications made by web page scripts. However, even the latest releases of Firefox continue to suffer from bugs in assigning wrappers, thus allowing privileged scripts to interact with tampered DOM methods and properties [6].

#### 3.2.4 Privileged Reference Leaks

Figure 5(a) shows code that exploits a reference leak vulnerability in the QueryInterface XPCOM object. A flaw in the XPConnect code for setting up safety wrappers for native objects inadvertently sets a privileged object as the prototype of the safety wrapper for QueryInterface in untrusted namespaces. Malicious code can use this leak to reach the global

[2]This Firefox-specific object has been deprecated since Firefox 3.0, presumably due to security risk.

```
var leaked =
  QueryInterface.__proto__.__parent__;
var cid = {equals:  Script(payload)};
leaked.foo.getClassObject(cid);
```

(a) Simplified exploit code.

```
var foo = {
  getClassObject:  function(aCID) {
    if (aCID.equals(value))
      return this._objects[key];
  }
};
```

(b) Simplified target code.

Figure 5: Exploit and target code for Bug 294795.

object of a privileged namespace. The exploit calls the script method `foo.getClassObject` in the privileged namespace with a specially-crafted argument to carry out a luring attack.

The `getClassObject` method shown in Figure 5(b) relies on namespace isolation and thus expects to be called from other privileged functions with safe arguments. However, when it calls the `equals` method of its `aCID` parameter, it inadvertently invokes the Script object defined by the attacker, executing it with full privileges.

#### 3.2.5 Loading Privileged URIs

There are also attacks that use a combination of a bug that allows unprivileged pages to load higher privilege documents (e.g., "chrome" protocol URIs) and a cross-site scripting (XSS) bug to inject their own scripts into these pages. Bug 306261 allowed untrusted pages to bypass restrictions on loading privileged URIs of the "about" protocol (which allows setting browser configuration values) by using a malformed URI. We do not address XSS bugs or violations of URI loading policies, but our system is able to detect this category of attacks because it leads to code injection.

### 3.3 Comparison With Memory Safety

JavaScript extensions have many clear benefits, but they suffer from risks posed by these four classes of vulnerabilities. As a result, Firefox users have been victims of real-world privilege escalation attacks and the Firefox bug database shows that the incidence rate for these types of vulnerabilities is comparable to memory-safety vulnerabilities (more on this in Section 5.1).

At first, this may seem counter-intuitive: components written in a memory-safe, interpreted language should be more secure than their native equivalents. This intuition may be true in single-principal applications, but Firefox must execute JavaScript from multiple principals concurrently and must arbitrate over many possible interactions, which raises the specter of bugs leading to privilege escalation attacks.

In fact, the classes of vulnerabilities we found for the multi-principal Firefox script environment are similar to memory-safety vulnerabilities found in single-principal native code. The code compilation vulnerabilities are not unlike buffer overflows: data is executed as code, allowing for arbitrary code execution. The luring vulnerabilities allow attackers to call existing functions of their choosing, similar to return-to-libc attacks [5].

## 4 Approach

Script-based extensibility in the Firefox web browser is a powerful feature and is highly valued by its users. However, it leads to privilege escalation vulnerabilities precisely because of the dynamic and flexible nature of the script language used to implement the extensions. The language features allow leveraging browser bugs or vulnerable extensions to confuse the browser into assigning wrong principals to code, thus bypassing stack inspection.

Privilege escalation vulnerabilities also arise because Firefox's implementation of one-way namespace isolation is inherently error prone. The browser fully trusts script extensions, but these scripts can interact with data from unprivileged sources in unsafe ways, compromising namespace isolation. One-way namespace isolation will not disappear from extensible browser architectures, as extensions will always need to read and modify untrusted web pages. One method of improving the security of one-way namespace isolation is to provide stronger isolation guarantees. For example, Google Chrome [10] divides an extension into separate processes, one for for accessing privileged interfaces, and another for interacting with untrusted web pages, while only allowing IPC between the two processes. This architecture requires increased implementation effort from the extension developer and is completely incompatible with the Firefox extension model.

Instead, our solution is to use tainting to augment the browser's security mechanisms. We use tainting because it helps detect when untrusted content can affect privileged code. Furthermore, it is fully compatible with the current Firefox extension model. Unfortunately, many tainting-based systems suffer from endemic false alarms and thus are unusable in practice [18]. In this section, we show that our tainting-based solution, while being conceptually simple, is well-suited for the browser's security model because namespace isolation already provides a security barrier between the taint sources in content namespaces and privileged code in extension namespaces.

### 4.1 Threat Model

We define a privilege escalation attack as tainted data executing as privileged code. Tainted data is executed as privileged code if it is compiled into script byte code tagged with the wrong principals, or if tainted data is used as a reference to execute privileged code. Both scenarios lead to a failure of the browser's security mechanism for guarding access to sensitive interfaces, allowing untrusted web pages to gain the ability to modify the host system.

We add security checks and augment stack inspection to look for the characteristic signature of privilege escalation attacks. To do so, we rely on the memory safety of the browser as well as the browser's ability to correctly assign a principal to a web page when it is *first* loaded, before any web page scripts begin executing. Assigning this principal is straightforward as it only depends on the web page's URI. We do not depend on the correctness of the rest of the code that assigns principals, or code that interprets principals. Instead, we "second guess" browser security code by auditing its security decisions with the additional taint status information.

### 4.2 Tainting

We consider all documents fetched from remote sources or local documents opened with the "file" protocol as untrusted and taint them because the browser does not assign them a privileged princi-

pal. When documents are loaded into the browser, they are parsed into a tree of native DOM objects, representing individual markup elements and their attributes. All nodes of the tree are individually marked tainted, including the text of any scripts defined inside the document, such as in event handlers or in SCRIPT tags, and taints are tracked separately for each attribute of a DOM element.

Our tainting system uses *different* policies based on the privilege level of the executing script. Unprivileged code is completely untrusted and may be malicious, so we must unconditionally taint all script variables created or modified by executing scripts originating from untrusted (tainted) documents. For privileged scripts, we use standard taint propagation rules that mark the output of JavaScript instructions as tainted when the instruction inputs are tainted. Tainting allows us to mark and track the influence of untrusted code throughout the browser.

Tainting systems can suffer from excessive false alarms when using control-dependent tainting. Control-dependent tainting taints the output of any code whose execution depends on tainted data. For example, all outputs of an if-branch would be tainted if the condition variable were tainted. Control dependence is necessary when the code processing the tainted data may itself be malicious. For example, detecting cross-domain information leaks requires accounting for implicit flows, since malicious web page scripts could leak information [19]. We do not use control-dependent tainting on the privileged side because we assume that the privileged scripts are trusted. We consider it highly unlikely that privileged script code would accidentally launder taints through control flow and then execute the laundered data as privileged code.

It is necessary to track taint both in the native code and inside the script interpreter. For example, when a new HTML document is loaded into a tab, privileged UI script code reads the tainted document's title property and sets it as the caption of the tab element. This requires taints from native DOM objects associated with the HTML document to propagate to script variables in the UI code and then back to DOM objects associated with the UI document. On the native side, we track the taint status of string properties of XPCOM objects. Tainting code in XPConnect taints any JavaScript references to unprivileged DOM elements and propagates taints between the XPCOM and JavaScript environments.

### 4.3 Attack Detection

We define a privilege escalation attack as tainted data executing as privileged code. We implement two classes of attack detectors to detect this condition: compilation detectors and invocation detectors. Compilation detectors ensure that tainted data is never compiled into byte code tagged with privileged principals, while invocation detectors monitor the stack for tainted references to function objects creating privileged frames. Compilation detectors map closely to code compilation vulnerabilities, while invocation detectors are best suited for preventing luring attacks.

#### 4.3.1 Compilation Detectors

We use compilation detectors as a proactive measure to prevent tainted data from being compiled to privileged byte code, even if it is never executed. These detectors are well suited for securing eval-like functions that compile strings into byte code, because the string's taint status informs these functions of the string's origin. These detectors allow defending against compilation bugs such as the wrong principal attack (see Section 3.2.2). If native XPCOM code compiles the strings, as in the URI code injection attack (see Section 3.2.1), or the XSS attacks (see Section 3.2.5), the detectors will use the taint status of XPCOM string objects to detect and prevent exploits. Our compilation detectors are placed before all calls to compilation functions, such as those defined by the JavaScript API.

#### 4.3.2 Invocation Detectors

Invocation detectors monitor script execution for situations where tainted references to script or native functions are invoked inside the interpreter and result in the creation of privileged stack frames. This policy catches luring attacks in which privileged scripts are tricked into invoking functions of the attacker's choice. It also detects when an unprivileged script uses a reference leak to directly call a privileged JavaScript function from an extension.

The invocation detectors vary depending on whether the invoked functions are scripted or native. Namespace isolation limits script functions to calling other script functions within the same namespace. Therefore, our detectors watch for namespace pollution, namely callers invoking tainted function references that result in a privileged callee stack frame, as in the luring attack (see Section 3.2.3). This detector is able to intercede before any function code is executed with elevated privileges.

For native functions, it is not as straightforward to come up with a policy for detecting attacks. It can be perfectly safe for privileged scripts to invoke natively defined methods of tainted object references. For example, an extension script could call the native `toLowerCase` string method on a web page's title string. The reference to the title string will be tainted, and the function reference to the `toLowerCase` method will also be tainted because it is accessed as a method of a tainted string, but this operation should not raise a privilege escalation alert because, in and of itself, it does not represent a privilege escalation threat even if it is called from a privileged context. However, if the native function called through the tainted reference is a native XPCOM method that is only accessible to privileged callers, then a security violation needs to be raised as it indicates a luring attack.

Thus, it is important to know whether the native callee is sensitive and whether the caller will be interpreted as privileged. We get this information by letting the call proceed, and if it reaches XPConnect, the security manager establishes the sensitivity of the target XPCOM method or property and performs a stack inspection to determine the effective subject principal of the caller. We augment the security manager to signal an attack whenever it computes a privileged subject principal, but a tainted function reference is found on any stack frame during the stack walk.

#### 4.3.3 Reference Leaks

As demonstrated in Section 5, we can detect and stop the vast majority of proof-of-concept exploits in the Firefox bug database based on reference leaks. We achieve these results by detecting attempts to directly invoke or lure privileged code with our invocation detectors, as in the reference leak attack (see Section 3.2.4), and by detecting malicious attempts to compile tainted strings with our compilation detectors. However, we are unable to detect and prevent reference leaks. For example, in Figure 5(a), we cannot rely on the object reference's taint status to detect the privileged reference leak, because our tainting rules require that properties of tainted objects, such as QueryInterface, also be marked tainted.

Although we do not prevent reference leaks, attacks employing reference leaks will not be able to escape our tainting. Any data modified by untrusted scripts is still marked tainted, and invoking or compiling tainted data will trip the detectors. Therefore, attackers will not be able to mount a privilege escalation attack, in which untrusted data is executed as privileged code. At most, if the reference leak allows access to arbitrary global variables in the privileged namespace, attackers may be able to devise control dependent attacks and compromise the integrity of extension logic.

Barth et al. [11] propose a system for detecting reference leaks between different security origins. Although their work aims to prevent cross-origin attacks made possible by reference leaks, it could also be integrated with our system to detect reference leaks from privileged namespaces. We should note that reference leaks are not a requirement for mounting luring attacks. As previously described in section 3.1.2, the target of any luring attack can also be a call to an eval-like function (such as the Script object) or a reference to a sensitive method of an XPCOM object legitimately present in the content namespace.

#### 4.3.4 Unsafe XPCOM Arguments

We are currently conducting a study to determine the extent of this class of vulnerability. We plan to create a list of sensitive parameters of security-sensitive XPCOM interfaces known to the security manager to mitigate the threat of tainted XPCOM arguments. We would need to provide untainting functionality to allow privileged scripts to indicate that a tainted argument has been sanitized. Other systems, such as Saner [9], allow validating sanitization routines.

### 4.4 Implementation

In this section, we describe the implementation of our tainting system in the JavaScript interpreter and

the XPCOM classes and our attack detectors. In our system, we are most concerned about the taint status of strings and function references because privilege escalation attacks require either luring privileged code or compiling attacker strings. We chose not to use an existing system-level tainting solution because control dependent tainting is not required in our system and low-level tainting systems tend to produce a large number of false positives.

#### 4.4.1 JavaScript Interpreter

JavaScript tainting requires associating a notion of taint with each script variable. JavaScript variables can hold the values of primitive data types such as booleans and integers, or they can hold references to heap allocated data, such as objects, strings, and doubles (hereafter collectively referred to as "objects"). All accesses to object variables are done by reference. We transparently convert all tainted primitive variables to doubles (a reference type) so that our tainting code exclusively deals with reference types. For reasons which we will discuss shortly, we do not taint the actual heap object pointed to by the reference (e.g. the floating point value of a double variable), but instead we only ever taint the individual references (pointers). For example, it is possible to have both a tainted and an untainted reference (pointer) to the same string. Therefore, variables of all data types are tainted in the same way, i.e. by tainting individual references.

When we implemented our tainting system, we had a choice between associating taint status with objects or with references to objects. We believe that it is a mistake to associate taint with objects because objects can be safely shared across privileged and unprivileged namespaces. For example, if a string variable were to be defined in a privileged namespace and then assigned to a variable in an unprivileged namespace, and unprivileged code were then to copy it into another variable, the original reference and the copy should not have the same taint status although they reference the same heap object. The value of the copied variable was clearly influenced by untrusted code, whereas the original variable was not. Note that strings and doubles are immutable, so there is no risk of modification by untrusted code. In other words, whenever a string or a double is modified, a new object is created with the new value and the original remains unchanged. For mutable JavaScript objects, our policy is to taint individual property references when they are modified by untrusted code. If we were to taint by object instead of by reference, we would run the risk of excessive, unnecessary taint propagation. For example, if an extension stores a tainted value in a property of a commonly used object, the object itself would become tainted. Therefore, any existing fields or methods of the object would also become tainted without receiving any tainted data. Such tainting could lead to false positives. The most egregious example of such unnecessary taint proliferation occurs when an extension copies a tainted variable into its global namespace, which is itself an object. Tainting the global object instead of merely tainting the property reference would unnecessarily taint all existing variables in the trusted extension namespace.

Therefore, we implemented variable tainting by storing a taint bit inside each variable. Internally, JavaScript variables are a machine word with a few of the least significant bits reserved for a type tag used for dynamic typing. We set aside an extra bit in the type tag for the taint status. The upper bits of primitive variables contain the variable's value, while the upper bits of references contain a pointer to a memory-aligned heap object. A downside of our reference tainting approach is increased memory use because heap objects now have to align at bigger boundaries. Specifically, we can store half as many JavaScript objects within a single memory page. This may seem like a large overhead for our approach, but the heap-allocated data structures are very small because the data structures use *unaligned* pointers to point to their actual contents. For example, the aligned, heap-allocated string data structure consists of two member variables: the string length and a pointer to an *unaligned* character array stored elsewhere on the heap. In practice, we find the overhead is not significant because JavaScript heap memory accounts for only a small portion of the Firefox memory footprint. Empirical measurements confirm that the increase in Firefox's data resident set size is less than 10% in everyday browsing, even on JavaScript-heavy sites such as GMail.

We added code to propagate taint between the inputs and outputs of each of the 154 opcodes in the JavaScript interpreter as well as code to unconditionally taint all outputs produced by unprivileged scripts. In addition to the aforementioned data types, scripts can also make use of a number of built-in objects and top-level properties and functions defined by the JavaScript language. Some built-in objects provide more advanced data types such as the "Date" and "Array" objects, while other built-ins provide utility functionality such as the "Math" object and the "encodeURI" function. Instead of painstakingly modifying each of these methods and functions individually to propagate taints, we conservatively taint the return values from any built-in function or method if any supplied arguments are tainted. For example, the returned values from `Math.sqrt(X)` or `encodeURI(X)` will be tainted if `X` is tainted. Finally, we had to make a few manual changes in the interpreter code to prevent loss of taint. For example, object references were sometimes converted into raw pointers and then the same raw pointers were converted back into object references without restoring the taint bit in the type tag.

#### 4.4.2 XPCOM

We track the taint status of string objects in the XPCOM code because it is possible for native and interpreter code to compile strings into attack code. We also pay special attention to tracking taint in DOM string properties as these properties are the initial taint source and a very common taint sink.

We have borrowed the XPCOM string-tainting implementation from Vogt et al. [19]. This implementation adds taint flags to XPCOM string classes and modifies string class methods to preserve taint. We extended it to more string classes and made a small number of manual changes to account for the taint laundering that occurs in the code base when raw string pointers are extracted from string objects and used to create new string objects.

The XPCOM implementations of markup elements, representing the contents of the browser UI and web pages, do not store all their string properties within XPCOM string classes. The string properties of these DOM elements are a significant source and propagation vector for tainted data, so we needed to associate each string property of a DOM element with a taint status. To this end, we modified a small number of base classes from which DOM elements of all types are derived. DOM classes redirect calls to get or set individual properties to a handful of methods in these base classes, allowing us to add taint-propagation behavior and to automatically taint string properties of elements in unprivileged documents.

Adding taint tracking for every type of XPCOM property is difficult because there is no elegant way to associate taint status with primitive data types in the native XPCOM code. However, it is straightforward to taint all script references to unprivileged DOM objects. We added a taint bit to the "wrappers" used to reflect XPCOM objects into the JavaScript environment as well as the wrappers used to reflect JavaScript objects into XPCOM code. The first time XPConnect is asked to reflect a given object between the two environments, it creates a new wrapper object in the destination environment. For wrappers around XPCOM objects, we alter the wrapper creation process to check whether the wrapped object is a DOM node and if so, if it belongs to an unprivileged document. When the wrapper is placed in a JavaScript namespace, we make sure its object reference is tainted. The tainting rules in the interpreter automatically taint the values obtained from reading tainted objects' properties, effectively tainting all string and non-string properties of unprivileged DOM elements. Similarly, when a JavaScript object or function reference is wrapped for the XPCOM environment (e.g., a JavaScript callback function), we make sure its taint status is preserved and therefore propagated during a property read or a function call.

#### 4.4.3 Attack Detectors

Once we determined the detection policies described in sections 4.3.1 and 4.3.2, implementation of the attack detectors became straightforward. The compilation detector code was added to the native functions that turn strings into bytecode (such as "eval"), while the invocation detector code was added to the code that implements JavaScript function calls. The only challenge was in finding the appropriate sites to install the detectors so that all JavaScript compilation and function invocations could be audited. The detectors had to be close enough to the low-level compilation and invocation

code to intercept all the relevant call paths, but at the same time sufficiently high-level to easily retrieve principals and taint status.

## 5 Evaluation

We have implemented the approach described above in the Firefox browser. In this section, we evaluate our system by demonstrating its effectiveness against privilege escalation attacks. We start by showing how well it prevents attacks on known Firefox vulnerabilities. These vulnerabilities are documented in Firefox's Bugzilla bug database, which provides detailed security reports, proof-of-concept exploits and any available bug fixes. Next, we show that our system has minimal impact on normal usage by evaluating any false alarms that are raised and the performance overhead.

We evaluated against proof-of-concept attacks from Mozilla's bug database because the vulnerabilities are well cataloged and the proof of concept attacks are readily available. Most extension authors do not invest as much effort as Mozilla into documenting security issues in their code, thus making it difficult to evaluate our system against attacks on specific extensions. However, the same vulnerabilities could be leveraged against extensions.

We have implemented our system on Firefox version 1.0.0, which we use for all the experiments. We chose this version because it has the largest number of known privilege escalation bugs, allowing more extensive testing of our system. Also, the Firefox security team has a policy of embargoing reports for recent vulnerabilities, except for exploits already available in the wild. As a result, recent versions of Firefox have far fewer available privilege escalation exploits. For example, as of the end of 2009, the current version of Firefox (v3.5) has several privileged escalation vulnerabilities as shown below but no publicly available exploits for them. We plan to port our system and evaluate our results for newer versions of Firefox as exploits become available in the bug database.

### 5.1 Vulnerability Coverage

Table 1 shows the continuing threat posed by privilege escalation (PE) vulnerabilities in the Firefox browser. This table shows the total number of critical vulnerabilities and the number of critical PE vulnerabilities in the various major versions of the browser. The last column shows the percentage of PE vulnerabilities. Most PE vulnerabilities are generally classified as critical, and thus we do not show the statistics for non-critical vulnerabilities. Table 1 shows that PE vulnerabilities comprise 2/3 of all critical Firefox 1.0 vulnerabilities. All other versions continually have about 1/3 PE vulnerabilities. The main reason is that Firefox 1.5 implements safety wrappers that limit the opportunities for unsafe interactions between privileged code and web content, as described in Section 3.2.4.

| Firefox | Critical | Critical PE | % |
|---|---|---|---|
| Version 1.0 | 27 | 18 | 67 |
| Version 1.5 | 44 | 13 | 30 |
| Version 2.0 | 43 | 16 | 37 |
| Version 3.0 | 30 | 8 | 27 |

Table 1: Vulnerability Statistics.

Table 2 shows all the 19 privilege escalation advisories affecting Firefox 1.0.0, with some advisories containing multiple bug reports. Note that there are 26 such advisories in Firefox 1.0 (of which 18 are critical as shown in Table 1), but the other seven do not run on Firefox 1.0.0 and so we are unable to reproduce them. We were unable to test our system against 5 out of the 19 advisories because exploits were not available for them. The last column shows the types of vulnerabilities exploited in each advisory. For reference leaks, we also show whether the leak is leveraged to compile code (C) with the wrong principals or execute a luring attack (L).

Our system guards against 13 out of the 14 vulnerabilities described in the advisories. We do not detect an attack on the vulnerability in advisory #6. In this attack, an untrusted HTML string is parsed by the HTML parser to generate new HTML elements in a privileged document. Currently, we lose taint because we have not implemented taint propagation within the HTML parser.

### 5.2 False Positive Evaluation

We also tested our system by installing the top 10 most popular extensions that were available for Firefox 1.0.0, and then we manually browsed the Web. These extensions are Adblock Plus, FoxyTunes, NoScript, Forecastfox, Add N Edit Cookies, PDF Download, StumbleUpon, 1-Click Weather,

| # | Advisory | Advisory Name | Type of Vulnerability | Detection |
|---|---|---|---|---|
| 1 | 2006-25 | Privilege escalation through Print Preview | Compilation | Yes |
| 2 | 2006-16 | Accessing XBL compilation scope via valueOf.call() | Leak (C) | Yes |
| 3 | 2006-15 | Privilege escalation using a JavaScript function's cloned parent | Leak (C) | Yes |
| 4 | 2006-14 | Privilege escalation via XBL.method.eval | Leak (C) | Yes |
| 5 | 2005-56 | Code execution through shared function objects | Leak (C), Leak (L) | Yes |
| 6 | 2005-49 | Script injection from Firefox sidebar panel using data:// | Compilation | No |
| 7 | 2005-44 | Privilege escalation via non-DOM property overrides | Luring | Yes |
| 8 | 2005-43 | "Wrapped" javascript: URLs bypass security checks | Compilation | Yes |
| 9 | 2005-41 | Privilege escalation via DOM property overrides | Luring | Yes |
| 10 | 2005-39 | Arbitrary code execution from Firefox sidebar panel II | Compilation | Yes |
| 11 | 2005-37 | Code execution through javascript: favicons | Compilation | Yes |
| 12 | 2005-35 | Showing blocked javascript: pop-up uses wrong privilege context | Compilation | Yes |
| 13 | 2005-31 | Arbitrary code execution from Firefox sidebar panel | Compilation | Yes |
| 14 | 2005-12 | javascript: Livefeed bookmarks can steal private data | Compilation | Yes |
| Embargoed, or exploit not available | | | | |
| 15 | 2006-24 | Privilege escalation using crypto.generateCRMFRequest | N/A | N/A |
| 16 | 2006-05 | Localstore.rdf XML injection through XULDocument.persist() | N/A | N/A |
| 17 | 2005-58 | Firefox 1.0.7 / Mozilla Suite 1.7.12 Vulnerability Fixes | N/A | N/A |
| 18 | 2005-45 | Content-generated event vulnerabilities | N/A | N/A |
| 19 | 2005-27 | Plugins can be used to load privileged content | N/A | N/A |

Table 2: Vulnerability Coverage.

MR Tech Toolkit and FLST. A user, who is not associated with the project, browsed the Web for 5 hours, specifically visiting the top 100 most heavily visited web sites, as ranked by Alexa [2]. The user interacted extensively both with the web sites as well as with the extensions (e.g., directly invoking extension functionality by setting preferences).

The user's testing caused one alarm. This alarm was caused by Forecastfox, which displays the current weather forecast for a city of the user's choice. When a user searches for his city while setting his preferences, Forecastfox queries `accuweather.com` for cities matching the search string. When the user selects his city from the search results, Forecastfox concatenates several strings together including the full city name fetched from the web site and `eval`'s this expression to set the city option. Since the city name string originates from an untrusted web page, and the expression is evaluated in a privileged context, the alarm is raised. This code is unsafe because if the web site were compromised, the browsers of all Forecastfox users could be exploited. After seeing this alarm, we researched and found that Forecastfox for Firefox 3.0 has removed the `eval` statement.

We also performed automated testing by writing a Web crawler extension for Firefox. The crawler extension takes as input a list of web sites to visit and directs Firefox to load any HTML or JavaScript links found in the web site in depth-first order and interacts with each loaded page in Firefox to mimic the behavior of a human user. On each page, the crawler chooses multiple events to send to the page (e.g. mouse clicks, key strokes) and fills out and submits any HTML forms. The crawler exercises the JavaScript in the browser UI by performing one of several scripted GUI actions such as viewing the web page's HTML source code. We also installed AdBlock and Flashblock extensions and had the crawler randomly add and remove AdBlock filters on each page visited. The full crawler test visited 100 pages from each website in the Alexa Top 200.

The automated testing resulted in the discovery of one false positive, triggered by selecting "Page Source" from Firefox's "View" menu. The offending UI JavaScript retrieves a (tainted) reference to a window object from the content names-

pace. The window object implements multiple interfaces and some of these are sensitive interfaces inaccessible to web page scripts. The UI script casts the reference to the window object to a sensitive interface, further propagating the taint. When the privileged code calls a sensitive method of this interface through the tainted reference, our detectors flag it as a luring attack. This is not likely an exploitable vulnerability, but it would be safer if privileged JavaScript obtained references to sensitive interfaces without going through a content namespace.

While our testing is limited to heavily visited web sites, we believe that our system will not generate many false positives with other web sites. We find that privileged scripts are careful when operating on untrusted data and they are selective about the strings they compile in their privileged context (i.e., compilation false positives). Second, namespace isolation works well enough in non-malicious environments, and thus it is difficult for privileged function references to become tainted (i.e., luring false positives). Similarly, web pages don't expect to have access to privileged references and thus are unlikely to access them legitimately (i.e., reference leak false positives).

### 5.3 Performance

During regular browsing, we did not notice any degradation in page load times or responsiveness. We also conducted experiments to quantify the performance overhead of our system. We ran the Dromaeo JavaScript Tests and the DOM Core Tests from Mozilla's performance test suite [3]. These tests are micro-benchmarks that measure 1) the performance of basic operations of the script interpreter, and 2) the performance of common DOM operations. Our experiments were run on Ubuntu 8.04 Linux on an Intel Core 2 Duo 2.4 GHz processor, with 2 GB of memory. Our browser had 28% overhead for the JavaScript tests and 32% overhead for the DOM tests. Although the overhead witnessed in these micro-benchmarks does not visibly influence the browsing experience, the overhead may become an impediment to the adoption of our system at a time when JavaScript performance is becoming a competitive feature for modern browsers. One possible research direction would be to investigate how to efficiently integrate our tainting system with the just-in-time compilation systems present in modern JavaScript engines.

### 5.4 Security Analysis

Our system effectively detects nearly all available proof-of-concept attacks with few false positives. Admittedly, these proof-of-concept attacks were not designed with our detection system in mind. In order to defeat our defenses, an attacker would need to find a means of removing taint from untrusted objects. It would be difficult to remove taint in the JavaScript interpreter as the tainting rules are straightforward. The most likely target for laundering taint would be the native XPCOM methods.

One possible way for the browser to lose taint is to store tainted objects outside the browser. For example, if a user saves a malicious URL string from a web page as a bookmark, the bookmark is stored in a bookmarks file and the URI's taint is no longer present when the browser is restarted. A second, more involved method may be to launder taint through XPCOM method arguments. The attack begins by tricking an extension into passing a tainted, privileged object (a luring target) to an XPCOM function. If this function then natively calls a privileged native method of the tainted argument, our system would not detect this as a luring attack. This is because the extension JavaScript did not directly invoke a privileged method through a tainted reference. Similarly, if an XPCOM function were to accept a tainted object as an argument but then return a different, but related untainted object, it may be accurate to say the taint was laundered. Note that in these examples, the arguments and return values could not be strings as taint is always propagated during XPCOM string operations.

Although laundering taint is theoretically possible within our system, our system greatly raises the bar for potential attackers. The attackers now not only need to find a privilege escalation vulnerability in the browser, they also require extension JavaScript that interacts with specific XPCOM methods in such a way as to launder taint from crucial attack variables.



## 6 Related Work

This work focuses on securely executing untrusted scripts by using taint-based stack inspection. Stack inspection is widely used by modern component-based systems, such as Java and Microsoft .NET Common Language Runtime, to ensure that remote code is sufficiently authorized to perform a security-sensitive operation. Wallach et al. [20] provide instructive background on stack inspection.

Taint analysis helps determine whether untrusted data may influence data that is trusted by the system. Newsome and Song [16] use dynamic taint analysis to taint data originating or derived from untrusted network sources. An attack is detected when tainted data is used in a dangerous way, such as overwriting a return address. We use a similar approach to ensure that dirty data is not executed in a trusted context. Vogt et al. [19] use script tainting in a browser to track sensitive browser data, such as browser cookies or the URLs of visited pages.

The same origin policy is the basic sandboxing method used by web browsers. An effective method for implementing the same origin policy is script accenting [12], which uses simple XOR encryption to ensure that code is loaded and run, and data is created and accessed, by the same principal. Several recent projects [22, 17] attempt to enforce the same origin policy by separating different origins into different processes. In order to adopt this architecture, the extension model needs to be redesigned to accommodate extensions' interactions with pages from different principals [10]. The same origin policy is too strict for mashup web applications. For such applications, Mashup OS provides abstractions to allow limited communication while protecting the different principals associated with mashup content [21]. Interestingly, Mashup OS introduces the same set of problems as privileged extensions interacting with untrusted content and thus would benefit from our solution.

In concurrent work, Barth et al [10] propose a new browser extension model for Google Chrome. Extensions and web page scripts are isolated using processes and "isolated worlds" so that they never exchange JavaScript pointers. This architecture raises the bar for perpetrating a successful privilege escalation attack as multiple components now need to be compromised. Their design has obvious advantages, but the threat of privilege escalation attacks has not been completely eliminated. For example, Google recently fixed a vulnerability that incorrectly allowed JavaScript to be executed in the context of a Chrome extension [7].

Since browser extensions typically run with unrestricted privileges, a malicious extension can serve as a powerful attack vector. Louw et al. [15] propose access control for limiting extension privileges. For example, certain extensions may not be allowed access to the password manager. Dhawan and Ganapathy [13] propose adding an information-flow tracking system to Firefox to assist in determining whether a JavaScript extension maliciously compromises browser confidentiality or integrity. Although we are also interested in misuses of low-integrity data, their system is not an online attack detector and it requires human analysis.

Recent versions of Firefox use security wrappers (e.g., XPCNativeWrappers, XPCChromeObjectWrappers, etc.) to regulate interactions between JavaScript and XPCOM objects from different namespaces [8]. Unfortunately, implementation bugs in creating and manipulating wrappers are fairly common. Our system adds another layer of security on top of wrapper techniques by effectively second guessing wrapper security decisions.

## 7 Conclusion

Script-based privilege escalation attacks are a serious and recurring threat for extensible browsers such as Firefox. In this paper, we describe the pitfalls of script-based extensibility in Firefox and show that the privilege escalation vulnerabilities are similar to arbitrary code injection and execution control vulnerabilities found in unsafe code. Then, we propose a tainting-based system that specifically targets each class of vulnerability. We implemented such a system for the Firefox 1.0 browser and our evaluation shows that it detects the vast majority of attacks in the Firefox bug database with almost no false alarms and moderate overhead.

Our vulnerability classification and our proposed defense system are inevitably linked to the Firefox browser. However, one-way namespace isolation must exist in browser extension architectures because extensions need access to restricted APIs

and they also need to read and modify untrusted web pages. As such, we expect our analysis and results to be applicable to other script-extensible browsers.We plan to test the generality of our vulnerability classification and defenses against other browsers, especially Google Chrome as it also provides powerful script extension functionality.

## Acknowledgments

We would like to thank our shepherd, Helen Wang, and the anonymous reviews for their insightful comments on the paper.

## References

[1] Adblock. http://en.wikipedia.org/wiki/Adblock.

[2] Alexa the web information company. http://www.alexa.com.

[3] Dromaeo JavaScript performance test suite. https://wiki.mozilla.org/Dromaeo.

[4] Greasemonkey. http://en.wikipedia.org/wiki/Greasemonkey.

[5] Return-to-libc attack. http://en.wikipedia.org/wiki/Return-to-libc_attack.

[6] setTimeout loses XPCNativeWrappers, July 2009. http://www.mozilla.org/security/announce/2009/mfsa2009-39.html.

[7] Incorrect execution of JavaScript in the extension context, May 2010. http://googlechromereleases.blogspot.com/2010/05/stable-channel-update.html.

[8] XPConnect wrappers, May 2010. https://developer.mozilla.org/en/XPConnect_wrappers.

[9] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[10] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[11] A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *Proceedings of the USENIX Security Symposium*, Aug. 2009.

[12] S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2–11, 2007.

[13] M. Dhawan and V. Ganapathy. Analyzing information flow in Javascript-based browser extensions. In *Proceedings of the Annual Computer Security Applications Conference*, 2010.

[14] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 339–354, 2008.

[15] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, Aug. 2008.

[16] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2005.

[17] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the EuroSys conference*, 2009.

[18] A. Slowinska and H. Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of the EuroSys conference*, Apr. 2009.

[19] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[20] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 116–128, 1997.

[21] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 1–16, 2007.

[22] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the USENIX Security Symposium*, 2009.

[23] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 79–93, 2009.



# AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements

Mike Ter Louw Karthik Thotta Ganesh V.N. Venkatakrishnan

*Department of Computer Science*
*University of Illinois at Chicago*

## Abstract

Web publishers frequently integrate third-party advertisements into web pages that also contain sensitive publisher data and end-user personal data. This practice exposes sensitive page content to confidentiality and integrity attacks launched by advertisements. In this paper, we propose a novel framework for addressing security threats posed by third-party advertisements. The heart of our framework is an innovative isolation mechanism that enables publishers to transparently interpose between advertisements and end users. The mechanism supports fine-grained policy specification and enforcement, and does not affect the user experience of interactive ads. Evaluation of our framework suggests compatibility with several mainstream ad networks, security from many threats from advertisements and acceptable performance overheads.

## 1 Introduction

On September 13, 2009, readers of the New York Times home web page were greeted by an animated image of a fake virus scan. Amidst widespread confusion, NY Times clarified the situation in an article [48], explaining the source of the rogue anti-virus attack was one of its advertising partners. Just two months prior, members of social web site Facebook were presented with advertisements (henceforth, "ads") deceptively portraying private images of their family and friends [38]. Facebook responded in an article [42] blaming advertisers for violating policy terms governing the use of personal images.

Publishers of online ads (like the NY Times and Facebook) face two serious challenges. They must ensure ads will neither violate the integrity of publisher web pages (as occurred with NY Times), nor breach confidentiality of user data present on publisher web pages (as occurred with Facebook). Ads are often tightly integrated into publisher web pages, and therefore must coexist with high integrity content and sensitive information. Typically, ad content is dynamically fetched from ad networks (e.g., Google AdSense) by the user's browser, leaving little opportunity for publishers to inspect and approve ads before the ads are rendered.

Online advertising is currently a lucrative market, expected to hit the US$50 billion mark in the U.S. during 2011 [52]. For many publishers, online advertising is an economic necessity. However, publishers have few resources enabling them to enforce integrity and confidentiality policies on ads. One common approach is for ad networks to screen each ad for potential attacks. This *passive* approach simply shifts the burden of protection from publisher to ad network. To enforce compliance, publishers must use out-of-band mechanisms (e.g., legal agreements), which leave the publisher vulnerable to any gaps in the ad network's screening strategy. Rogue ads may slip through and cause damage, as in the above, high profile examples.

Due to the dangers of rogue ads, publishers are in great need of an *active*, technological approach to protect themselves and their end users. Therefore, in this paper we confront the problem of rogue ads from a publisher-centric perspective. At a basic level, a publisher is a web application that includes dynamically sourced content from an ad network in its output. Our objective is to empower this web application to serve ads from mainstream ad networks, while protecting its end users from several threats posed by rogue ads.

### 1.1 Contributions

In this paper, we present ADJAIL, a framework that aids web applications to support rendering of ads from mainstream ad networks without compromising publisher security. Our framework achieves this protection by applying policy-based constraints on ad content. There are five significant contributions of our approach:

1. *Confidentiality and integrity policy specification and enforcement.* We define a simple and intuitive policy specification language for publishers to specify several confidentiality and integrity policies on advertisements at a fine-grained level. We provide a novel and con-

ceptually simple policy enforcement mechanism that offers principled security guarantees.

2. *Compatibility with ad network targeting algorithms.* Ad networks use targeting algorithms to select which ads to display, based on several factors such as page context and user behavior. In many cases, these algorithms are implemented as scripts that analyze publisher content to select and fetch appropriate ads to be displayed. Our approach supports these targeting scripts, with the added benefit of restricting the targeting script's access to sensitive data.

3. *Compatibility with ad network billing operations.* Ad networks employ complex billing strategies based on several metrics, including ad *impressions* (number of times an ad is shown) and mouse clicks. Furthermore, ad networks have mechanisms for dealing with click fraud [2]. To remain transparent to billing and click-fraud detection mechanisms, our approach preserves impression and click metrics.

4. *Consistency in user experience.* Our approach does not affect the user experience in interacting with ads, for any change in the user experience (in terms of content, position and interactivity) may reduce the effectiveness of advertising. Furthermore, ADJAIL highlights the security trade-offs that are required for ensuring consistency in user experience for certain types of ads (such as inline text ads).

5. *Satisfaction of practical deployment requirements.* Publishers should not have to expend significant labor in adopting a new framework, as this may make adoption prohibitively expensive. Furthermore, publishers should be able to deploy a solution that does not require end users to install new client software (e.g., browsers, plug-ins, etc.) or make changes to their existing client software. Therefore, we offer a practical solution that is easy to adopt, and works on mainstream browsers in their default settings, without any modifications.

### 1.2 Overview

The crux of our approach is a novel policy enforcement strategy that can be employed by the publisher to interpose itself transparently between the ad network and end user. The enforcement strategy starts by fetching and executing ads in a hidden "sandbox" environment in the user's browser, thus shielding the end user and web application from many harmful effects.

In order to preserve the user experience, all ad user interface elements are then extracted from the sandbox and communicated back to the original page environment, as permitted by the publisher's policy. This step enables the user to see and interact with the ad as if no interposition happened. All user actions are communicated back to the sandbox, thus completing a two-way message conduit for synchronization. Our approach ensures transparency with regard to the number of ad clicks and impressions by interposing on the browser's Document Object Model to suppress extraneous HTTP requests.

We have built a prototype implementation of ADJAIL that supports specification and enforcement of fine-grained policies on ads sourced from leading ad networks. The prototype is designed to be compatible with several mainstream browsers including Google Chrome, Firefox, Internet Explorer (IE), Safari and Opera. One minor limitation of our implementation (but not of our architecture) is that it is not compatible with IE 7.x or below. However, the current ADJAIL prototype is compatible with IE 8.0 and later.

We evaluate ADJAIL on the dimensions of ad network compatibility, security, and performance overheads. Our compatibility evaluation tested ads from six mainstream ad networks. We find that ADJAIL provides excellent compatibility for most ads. We also demonstrate the strong protection offered by ADJAIL from many significant threats posed by online ads. In our experiments, the currently unoptimized ADJAIL prototype encountered at most a 1.69× slowdown in rendering ads.

The remainder of this paper is organized as follows: Section 2 provides the threat model, scope and related work. We provide the architecture and the main ideas behind ADJAIL in Section 3. Section 4 discusses the details in the implementation of ADJAIL. Our security, compatibility and performance evaluation appears in Section 5. In Section 6 we conclude.

## 2 Threat Model and Related Work

### 2.1 Threat model

Consider a publisher who wishes to carry ads on a *webmail* (Web-based email) application. We will use this as a running example throughout the paper to illustrate the various aspects of our framework. A screenshot from an actual webmail application we used in our evaluation appears in Figure 1. The top pane of the window presents the message list and the bottom pane presents the email message text. Four numbered advertisements also appear in the figure: (1) a banner ad that appears on top of the webmail page, (2) a skyscraper ad that appears as a sidebar, (3) an inline text ad that appears when the user's mouse hovers over an underlined word, and (4) a floating ad that overlays the image of a clock on the page.

These ads highlight two interesting challenges we need to overcome. First, the sidebar ad requires access to the email message text, which it mines to ascertain page context and select relevant ads for display (i.e., *contextual targeting*). The inline text ad also requires access to the message for contextual targeting and to integrate ads among the text. However, supporting these ads by providing access to the entire message carries the risk of exposing private content (e.g., email addresses) to the ad script. Second, the floating ad requires access to the real estate of the page to place the image of the clock over the message text. However, providing access to the page real estate enables an ad to overlay content over the entire page, which may interfere with trusted interface components.

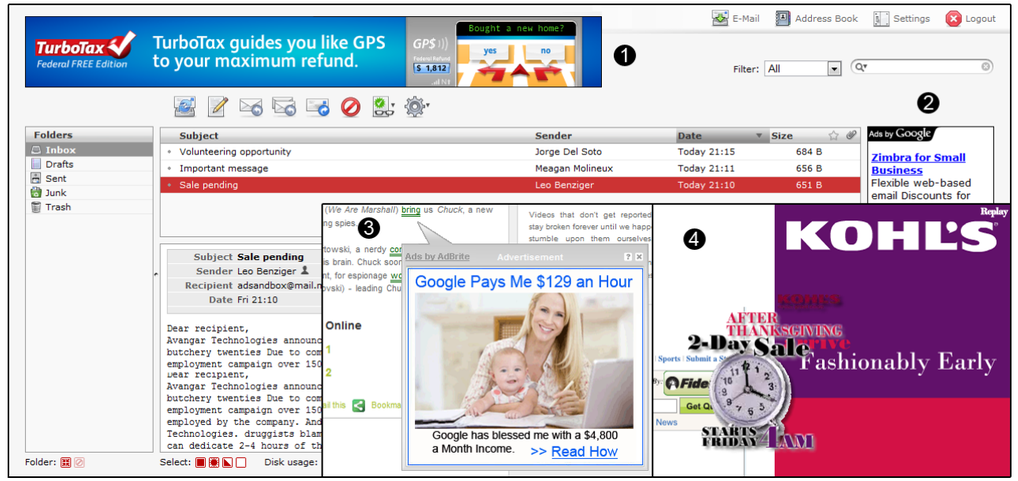



Figure 1: Samples of various ad types. A webmail application with (#1) banner and (#2) skyscraper ads. Also illustrated are (#3) an inline text ad and (#4) an floating ad.

These common examples illustrate how ads require non-trivial access to publisher content and the screen, and will not work if such access is denied. Also, in all of the examples above, the ad content is loaded and rendered by a third-party ad script (an ad script example appears in Figure 4a). Ad scripts are given full page access by default, and thus pose threats to the confidentiality and integrity of page content. Our goal is to support the non-trivial access required by these and many other typical forms of ads, while addressing the security concerns of executing third-party ad scripts.

### 2.2 Threat scope

Web applications that display third-party content on client browsers are exposed to a wide variety of threats. It is therefore important to clarify our threat model, specifically on the nature of protections that we offer and the threats that are outside the scope of this work.

**In-scope threats** The broad threats that we address in this work are those targeted by recent efforts in the Web standards community for content restrictions (e.g., Content Security Policy [32, 43]). These policies are specified by a website to restrict the capabilities of third-party scripts, specifically with reference to access and modification of first-party (site owned) content, as well as control over the screen. Policies can be negotiated between a publisher and its customers, or directly reflect the site security and privacy practices.

Our framework provides a means for specification and enforcement of such policies. For instance, in our webmail example, an integrity policy can be enforced such that email message content cannot be tampered with, but can still be read (for contextual targeting of ads). Publishers may also choose to restrict where ads can appear on the page.

Publishers can also use our framework to enforce policies about confidentiality of content. For instance, a publisher can enforce a policy that mail headers and email "address books" (containing private email addresses) cannot be read by ads. For the Facebook attack in §1, a policy specifying confidentiality of user images, combined with our enforcement mechanism, would have prevented the attack.

**Out-of-scope threats** Many security threats posed by ads (and other third party content) have been identified by the security community. Recently, there has been intense research in this area which can complement our approach for protection against specific attacks. In particular, our work does not address the threats listed below. In this section we omit threats for which publishers can readily deploy strong protection (e.g., cross-site request forgeries).

1. *Browser security bugs.* We do not address browser vulnerabilities such as drive-by-downloads [49, 36, 5], attacks launched through plug-ins [24], vulnerabilities in image rendering [23] and so on.

2. *Opaque content.* Our approach leverages web content introspection capabilities of JavaScript, and is therefore most capable of enforcing fine-grained control where such transparency is available. Although our approach provides coarse-grained confidentiality and integrity protection from opaque content (e.g., Flash), the many possible attack vectors from these binary formats require special treatment [13].

3. *Frame busting & navigation attacks.* These are difficult attacks for any dynamic policy enforcement mechanism to prevent, due to the limited API exposed by browsers. A detailed discussion of protection measures against frame busting has been explored [39] and could be used to enhance our approach.

4. *Behavior tracking attacks.* These are attacks that track a user across multiple sites and sessions through use of cookies. These could be addressed by users choosing restrictive cookie policies, though such policies may interfere with the functionality of some web sites.

5. *Attacks through side channels.* Sites can track users through side channels, such as the cache timing channel [11], the "visited links" feature of browsers [19] and so on. It is difficult to defend these vectors without browser customization, which is impractical for publishers to deploy.

### 2.3 Related Work

**Privacy and behavioral targeting** A few recent approaches have looked at the problem of addressing security issues in online advertising. Privads [15] and Adnostic [47] address this problem primarily from a user privacy perspective. They both rely on specialized, in-browser systems that support contextual placement of ads while preventing behavioral profiling of users. In contrast, our work mainly focuses on a different, publisher-centric problem of protecting confidentiality and integrity of publisher and user-owned content. Our work is also aimed at providing compatibility with existing ad networks and browsers.

**Restricting content languages** There have been a number of works [9, 6, 28, 29, 30, 12] in the area of JavaScript analysis that restrict content from ad networks to provide security protections. These works focus on limiting the JavaScript language features that untrusted scripts are allowed to use. The limitation is enforced statically by checking the untrusted script and ensuring it conforms to the language restrictions. Only those language features that are statically deterministic and amenable to analysis are allowed. Since much of the policy enforcement is done statically, these solutions typically have good runtime performance. In the cases of FBJS [9] and ADsafe [6], untrusted scripts are allowed to make calls to an access-controlled DOM (document object model) interface, which incurs some overhead but affords additional control. The cost in employing a restricted JavaScript subset is that ads authored by many advertisers may not conform to this subset, and therefore require re-development of ad script code. In contrast, ADJAIL neither imposes the burden of new languages nor places restrictions on JavaScript language features used in ad scripts. The only effort required from a publisher that incorporates ADJAIL is to specify policies that reflect site security practices.

**Code transformation approaches** Many recent approaches [37, 53, 22, 14, 34, 10, 35] have been pursued to transform untrusted JavaScript code to interpose runtime policy enforcement checks. These works cover the many diverse aspects by which third-party content may subvert policy enforcement checks. Since these works are aimed at general JavaScript security, they are not specialized to the problem of securing ads for publishers, where the main issue is ensuring transparent interposition. This is to avoid any conflict with ad targeting and billing strategies employed by ad networks. The recommended method of transforming JavaScript dynamically by a publisher involves using a proxy (e.g., for handling scripts sourced from an external URI). However, routing all ad script HTTP requests through a script-transformation proxy may appear suspicious to click-fraud detection mechanisms [2] employed by the ad network.

**Publisher-browser collaboration** An alternative approach is for a publisher to instruct a browser to enforce the publisher's policies on third-party content, leaving the enforcement entirely to the browser. This publisher-browser collaborative approach is a sound one in the long term to enforce a wide range of security policies as illustrated in BEEP [21], End-to-End Web Application Security [8], Content Security Policies [43] and ConScript [33]. The main positives of this approach are that it can enforce fine-grained policies with minimal overheads. The primary drawback is that today's browsers do not agree on a standard for publisher-browser collaboration, leaving a large void in near-term protection from malicious third-party content.

## 3 Architecture

Let us revisit our running example of a publisher who wishes to carry ads on a webmail application. Recall that the publisher embeds an ad network's JavaScript code within the HTML of the webmail page to enable ads. In the benign case, this JavaScript code scans the webmail user's email message body to find keywords for contextual ad targeting, then dynamically loads a relevant ad. For simplicity, we refer to the ad network's JavaScript and an advertiser's JavaScript (the latter loaded dynamically by the former) as *the ad script*. This section gives a high level overview of how we prevent the ad script from performing a variety of attacks against the publisher and end user.

Our approach is to initially confine the ad script to a hidden isolated environment. The hidden environment is locally and logically isolated [27, 44] as opposed to requiring additional physical and remote resources [31]. We then detect effects of the ad script that would normally be observable by the end user, had the script not been confined by our approach. These effects are replicated, subject to policy-based constraints, outside the isolated environment for the user to observe and interact with. User actions are then forwarded to the isolated environment to allow for a response by the ad script. Thus we facilitate a *controlled cycle* of interaction between the user and the advertisement, enabling dynamic ads while blocking several malicious behaviors.

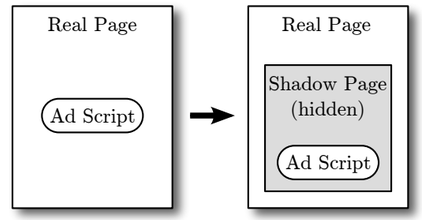
### 3.1 Ad confinement using shadow pages

As a basic policy, the publisher wants to ensure ad script does not access the publisher's private script data. If this policy is not enforced, ad script can read the sensitive `document.cookie` variable and leak its contents, enabling the recipient of the cookie to hijack the authenticated user's webmail session. Furthermore, ad script should not be allowed to read confidential user data from the page (e.g., email message headers and address book entries). Such data is normally accessible via the browser's document object model (DOM) script interfaces.

To enforce the publisher's policy, we leverage browser enforcement of the *same-origin policy* (SOP) [50], an access control mechanism available in all major JavaScript-enabled browsers. Web browsers enforce the SOP to prevent mutually distrusting web sites from accessing each other's JavaScript code and data. As a script instantiates code and data items, the browser places each item under the ownership of the script's origin principal. *Origin* principals are identified by the domain, protocol and port number components of the script's uniform resource identifier (URI). Whenever a script references code or data, both the script and item being accessed must be owned by the same origin, else access is denied.

To enforce the publisher's ad script policy, we begin by removing the ad script from the publisher's webmail page. Next, we embed a hidden `<iframe>` element in the page. This `<iframe>` has a different origin URI, thus invoking the browser's SOP and thereby imposing a code and data isolation barrier between the contents of the `<iframe>` and enclosing page. Finally, we add the ad script to the page contained in the hidden `<iframe>`. We refer to the hidden `<iframe>` page as the *shadow page*, and the enclosing webmail page as the *real page*. This transformation just described is depicted in Figure 2.

In the process of rendering the real page, the browser renders the shadow page, executing the ad script within. Our use of the SOP mechanism effectively relegates this ad script to an isolated execution environment. All access by ad script to code or data in the real page will be blocked due to enforcement of the SOP. Furthermore, the ad script can not retrieve confidential address book data via DOM interfaces, as access to those APIs are denied by SOP. We can say the publisher's basic policy is enforced, because (1) all such ad scripts are relocated to the shadow page, and (2) the browser correctly enforces the SOP.




(a) Before (b) After

Figure 2: Relocating the ad script to a hidden shadow page invokes the browser's *same-origin* policy for confinement.

### 3.2 Controlled user interaction with ads

Consider an ad script that loads a product image, or *banner*. Normally the banner appears on the real page, but since the ad script runs in the shadow page, the banner is rendered on the shadow page instead. Without further steps, the webmail user viewing the real page will never see this banner because the shadow page is hidden. We now describe how the user is able to interact with the shadow page ad by content mirroring (§3.2.1) and event forwarding (§3.2.2), subject to policy-based constraints (§3.2.3).

#### 3.2.1 Ad mirroring

A detailed view of the real and shadow pages that depicts mirroring of ad content is shown in Figure 3. We add Tunnel Script A to the shadow page that monitors page changes made by the ad script (①), and conveys those changes (②) to the real page via inter-origin message conduits [1, 20]. We add complementary Tunnel Script B to the real page that receives a list of shadow page changes and replicates their effects on the real page. Thus when ad script creates a banner image on the shadow page, Tunnel Script A sends a description of the banner to Tunnel Script B, which then creates the banner on the real page for the end user to see.

Special care is taken to prevent sending redundant HTTP requests to the ad server during the mirroring process, as such requests can interfere with an ad network's record keeping and billing operations. These details are discussed at depth in §4.3.2.

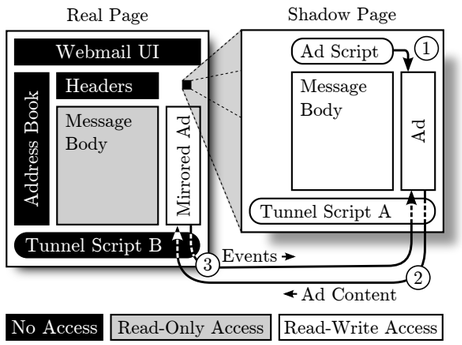



Figure 3: Overview of ADJAIL integrated with a webmail application. Ad script is given read-only access to email message body for contextual targeting purposes. Ad script can write to designated area to right of message body. Confidential data such as address book and mail headers are inaccessible to ad script.

#### 3.2.2 Event forwarding

Ads sometimes respond in complex ways to user generated events such as mouse movement and clicks. To facilitate this interaction, we capture events on mirrored ad content and forward these events (Figure 3, ③) to the shadow page for processing. For example, if the ad script registers an `onmousemove` event handler with the original banner image, we register our own (trusted) event handler on the mirrored banner image. Our handler listens for the mouse-move event and forwards it to the shadow page's banner via an inter-origin message. If the ad script responds to the mouse-move event by altering the banner or producing new ad content, these effects are replicated on the real page by our mirroring strategy outlined above.

#### 3.2.3 Ad policies

All messages sent between the real and shadow pages are mediated by our policy enforcement mechanism. This mechanism enforces policy rules which are specified by the publisher as annotations in the real page HTML. For the webmail example in Figure 3, the following access control policies are specified (shown in bold):

```
1 <div id="MessageBody"
2   policy="read-access: subtree;">
3   Message body text here...        </div>
4 <div id="Advertisement"
5   policy="write-access: subtree;"></div>
```

The policy in line 2 allows the ad script read-only access to the email message body. Read-only access is enforced by initially populating the shadow page with content from the real page (ref. Message Body regions in Figure 3). If ad script makes changes to read-only content, those changes are not mirrored back to the real page. Any attempts to mirror those changes to the real page message body (perhaps by a compromised Tunnel Script A) are denied.

The policy in line 5 permits the ad script write access to the sidebar on the right of the email message body. This is the region where the ad banner is to appear. When ad script creates content in the shadow page sidebar, this policy allows our mirroring logic to reproduce that content on the real page sidebar.

An implicit policy restriction on all mirrored content is that executable script code can not be written to the real page. To enforce this restriction, we only mirror items conforming to a configurable whitelist of static content types. Note this script injection threat is distinct from cross-site scripting (XSS), which the site can defend against using well-researched approaches (e.g., [46]).

The full policy language (detailed in §4.1) supports content restrictions to block Flash, deny the use of images (for text-only ads), restrict the size of ads, and more. These constraints can be tailored to the minimum compatibility requirements of individual ad networks, which we show in §5 can prevent attacks such as clickjacking [17].

Our policy enforcement mechanism is implemented on the real page as part of Tunnel Script B. As stated earlier, the ad script can not access the real page (including Tunnel Script B) due to SOP enforcement. Therefore ad script can not tamper with our policy enforcement mechanism.

## 4 Implementation

The implementation of ADJAIL is described in the context of a single webmail page with an embedded ad, which is integrated with our defense solution. We present the policy language used to restrict ads in §4.1. Then in §4.2 we describe how the real and shadow pages are constructed. §4.3 explains how we facilitate interaction between the two.

### 4.1 Policies

By default, ad script is given no access to any part of the real page unless granted by policies (i.e., *default-deny*). An implicit policy we always enforce is that ad script can not inject script code onto the real page, nor execute script code with privileges of the real page. We now describe in detail the individual permissions granted by policies, how policies are specified, and how multiple policies are combined to form a composite policy.

**Permissions** ADJAIL supports a basic set of permissions that control how ads appear on the real page and how ads can behave, summarized in Table 1. We define a *policy* as an assignment of values to each of the permissions. Our permissions have been designed iteratively by studying requirements of ads from several ad networks, and our results presented in §5 show the supported permissions can be composed to form useful advertisement policies.

The permissions `read-access` and `write-access` control what parts of the page ad script may read from or write to. Of particular interest is the `append` setting for `write-access`. This level of access allows ad script to add child content to an element, but neither read nor modify existing children of the element. Any appended children are automatically given a policy attribute set to `write-access: subtree;`. Some ads, such as the clock ad (#4) in Figure 1, require the `append` permission to add floating (i.e., absolutely positioned) content to the `<body>` element. In supporting these ads, we don't want to grant `subtree` write access to the `<body>` element, as that would enable a malicious ad to overwrite the entire page. Granting `append` access in this case is safer as it adheres to the principle of least privilege [40].

| Permission | Values | Description / Effects |
|---|---|---|
| `read-access` | `none`†*, `subtree` | Controls read access to element's attributes and children. |
| `write-access` | `none`†*, `append`, `subtree` | Controls write access to element's attributes and children. Append is not inherited. |
| `enable-images` | `deny`†*, `allow` | Enables support in the whitelist for `<img>` elements, CSS `background-image` and CSS `list-style-image` properties. |
| `enable-iframe` | `deny`†*, `allow` | Enables `<iframe>` elements in whitelist. |
| `enable-flash` | `deny`†*, `allow` | Enables `<object>` elements of type `application/x-shockwave-flash` in whitelist. |
| `max-height`, `max-width` | `0`*, *n* `%`, *n* `cm`, *n* `em`, *n* `ex`, *n* `in`, *n* `mm`, *n* `pc`, *n* `pt`, *n* `px`, `none`† | Sets maximum height / width of element to *n* units. Smaller dimensions are more restrictive. When composing values specified in incompatible units, most ancestral value wins. |
| `overflow` | `deny`†*, `allow` | Content can overflow boundary of containing element if allowed. |
| `link-target` | `blank`*, `top`, `any`† | Force targets of `<a>` elements to `_blank` or `_top`. Not forced if set to `any`. |

Table 1: Permissions that can be set in policy statements. *Most restrictive value. †Default value.

Part of our policy enforcement is a whitelist of HTML elements, attributes and CSS properties that ad script is allowed to write to the real page. Although this whitelist can be modified by the publisher at a low level, we support the following higher-order controls for tuning the whitelist. Ads are text-only by default; to enable images, the `enable-images` permission can be set to `allow`, thus expressing a publishers content restrictions policy on the use of third-party images. Another content restrictions permission is the `enable-flash` permission, that allows Flash ads to be displayed. Since our framework doesn't address security threats from opaque content such as Flash (§2.1), a publisher must exercise severe caution in enabling this permission. Also `<iframe>` elements can be allowed via `enable-iframe`. However, allowing `<iframe>` elements can facilitate attacks such as clickjacking [17] and drive-by downloads [36].

The `max-height`, `max-width` and `overflow` permissions control how the ad appears on the page. If an element's size surpasses the `max-width` or `max-height` dimension and the `overflow` permission is set to `deny`, then excess content is hidden. Otherwise the excess content will overlap other parts of the page. The overflow permission is useful because some ads consume a small area when not in use, but may overlap non-ad content when engaged by the user (e.g., expanding menus). Publishers may wish to disallow expanding ads because they can overlap trusted page content.

The `link-target` permission controls the HTML `target` attribute of all `<a>` elements (and `<form>` elements, if allowed by whitelist) in mirrored content. By setting this permission, the publisher can specify that activated links or submitted forms must be directed to a new browser tab / window (if set to `blank`), or directed to the tab / window hosting the real page (if set to `top`). Whether links open in the same or new window is often agreed to between the publisher and ad network. The `link-target` permission can be used to protect the publisher from ad script that mistakenly creates content that does not adhere to the agreed upon link behavior.

**Policy specification** The publisher can annotate any HTML element of the real page with a `policy` attribute. The `policy` attribute contains a set of statements, each terminated by a semicolon. Each statement specifies the value of a particular permission in the form, *`permission`*`: `*`value`*`;`. Acceptable values for *`permission`* and *`value`* are listed in Table 1.

Permissions granted in an element's `policy` attribute are inherited by descendants in the HTML document hierarchy. That is, the scope of a permission $\mathcal{P}$ is the HTML subtree rooted at the element whose `policy` attribute grants $\mathcal{P}$.

**Algorithm 1:** ComputePolicy( *targetElement* )

```
policy ← new Object();
WABeforeAppend ← undefined;
foreach element from root to targetElement do
    if policy[ "write-access" ] = "append" then
        policy[ "write-access" ] ← WABeforeAppend
    statements ←Parse( element.getAttribute( "policy" ));
    foreach stmt in statements do
        policy ← ComposePolicies( policy, stmt );
    if policy[ "write-access" ] ≠ "append" then
        WABeforeAppend ← policy[ "write-access" ];
foreach permission in all permissions do
    if permission is not defined in policy then
        policy[ permission ] ← GetDefaultValue( permission );
return policy;
```

**Policy composition** Multiple policy statements may assign different values to a single permission. This can occur within a single `policy` attribute or through inheritance. We resolve the ambiguity of multiple permission values through a composition process. The composition algorithm, given in Algorithm 1, takes a target element as input and derives an assignment of values to each of the permissions listed in Table 1.

We can describe the composition algorithm intuitively as follows. The effective value for a permission is the most restrictive value specified for that permission across all composed policy statements. That is, if a permission appears in multiple statements (either within an element's `policy` attribute or in separate inherited policies), we take the intersection of all specified values for the permission. After all statements have been composed, any permissions left unspecified are set to their most restrictive values.

To enhance usability we introduced three minor exceptions to the above. First, the `max-height` and `max-width` permissions default to their *least* restrictive value (i.e., `none`). We chose this default because a definitive maximum height and width will not be satisfactory for every type of ad. It is better for each publisher to explicitly declare these values if such restrictions are desired. The policy semantics is still *default-deny*, because write permissions must first be granted before restrictions on the size of written content can have any impact. For the same reasons, our second exception defaults `link-target` permission to its least restrictive value. The third exception is we prevent inheritance of `append` write permissions. This is important as `append` specifically does not grant access to existing children of an element; thus any existing children should not inherit the `append` permission.

(a)

```
<script type="text/javascript">
    google_ad_client = "pub-...";
    google_ad_width = 728;
    google_ad_height = 90;
    google_ad_format = "728x90_as";
    google_ad_type = "text";
</script>
<script type="text/javascript"
    src="http://pagead2.googlesyndi↵
        cation.com/pagead/show_ads.js"
></script>
```

(b)

```
<script type="text/javascript"
    src="AdJail.js"></script>
```

Figure 4: (a) Google AdSense ad script, removed from real page. (b) Tunnel Script B, added to real page.

### 4.2 Real and Shadow pages

The architecture of our implementation requires changes to the original web page (real page) and creation of a corresponding shadow page as described in §3.1. The shadow page is hosted on a web server having an origin different from the real page, thus the browser's same-origin policy ensures the shadow page by default has no access to the cookies, content or other data belonging to the real page. Deploying our implementation requires a publisher to configure their DNS and web server to support the shadow page origin domain. Care must be exercised in the selection of the shadow page domain (one for each advertiser) in order to ensure that there is no reuse or overlap of domains.

To facilitate voluntary communication between the two pages, we leverage the `window.postMessage()` browser API. `postMessage()` is an inter-origin frame communication mechanism that enables two collaborating frames to share data in a controlled way, even when SOP is in effect [1].

**Construction of the real page** The real page is a version of the publisher's original page modified in three ways. The first modification is to remove the ad script (Figure 4a). Second, we add the tunnel script (Figure 4b) to the end of the page. The third modification to the original page is annotation of HTML elements with policies, which we discussed at length in §4.1. Only two annotations, illustrated in §3.2.3, are required for the webmail example.

The real page tunnel script has an initialization routine that first scans the real page to find all elements with policies granting the following permissions: `read-access: subtree;`, `write-access: append;`, and `write-access: subtree;`. All matching elements are converted into models (i.e., JavaScript data structures) that will be sent to the shadow page in a later stage. Script nodes are omitted from models because we can not guarantee their semantics are preserved on the shadow page. An example model is shown in Figure 5, which models the readable Message Body `<div>` element in the webmail page (corresponding HTML given in §3.2.3).

```
{ nodeType: "ELEMENT_NODE",
  tagName: "div", syncId: 0,
  top: y, left: x, width: w, height: h,
  attributes: {
    id: "MessageBody",
    policy: "read-access: subtree;"
  },
  children: [
    {
      nodeType: "TEXT_NODE",
      nodeValue: "Message body text here..."
    }
  ],
  computedStyle: { ... }
}
```

Figure 5: Model of `MessageBody` element (as defined in §3.2.3) sent from real page to shadow page

Of the elements found in the initial scan, those with read permission are modeled by encoding (non-script) element attributes and readable child nodes into the model. The remaining elements (i.e., those having write access but no read access) are modeled as empty containers. That is, any attributes and child nodes are omitted from the model.

All elements with a policy annotation and their descendant elements are assigned a unique `syncId` attribute during initialization. The sync ID is used to match elements on the real page with their corresponding elements on the shadow page as content is kept synchronized between the two pages. As the final step of initialization, the tunnel script creates and embeds the hidden `<iframe>` element for the shadow page.

**Construction of the shadow page** The shadow page begins as a template web page containing only the tunnel script. As the template page is rendered, the shadow page tunnel script receives content models (described above) from the real page's tunnel script. The model data is sent as a character string in JSON [7] syntax via `postMessage()`. Once received by the shadow page, models are converted into HTML constructs using the browser's DOM interfaces. This results in a web page environment containing all the non-sensitive content and constructs of the real page, in which we will allow the ad script to execute.

To support ads (such as inline text ads) that appear or behave differently depending on where content is positioned, the shadow page is virtually sized to the dimensions of the real page, and content models are rendered in the same absolute position and size of their real page counterpart. Position and size information is depicted in Figure 5 as `top`, `left`, `width` and `height` properties. Throughout dynamic updates these attributes are kept synchronized by an approach given in §4.3.4.

Next, we install wrappers around several DOM API methods to interpose between the ad script and the DOM. Although ad script can circumvent our wrappers in Mozilla browsers by using the JavaScript `delete` operator [35], we do not rely on wrappers to enforce policies or security properties. Wrappers are used to monitor page updates and provide transparency with regard to the number of impressions generated by ads, which we discuss at length in §4.3.

**Default ad zone** Lastly, the ad script is embedded in the shadow page inside a container `<div>` element, which we automatically map to a corresponding `<div>` on the real page. We refer to these linked elements as the *default ad zone*. Automatic mapping is required because many ad scripts, such as Google AdSense, will not independently find and inject ads into the content imported from the real page. Rather they simply write ad content into the element containing the ad script. To support these ad scripts, the publisher indicates the default ad zone element on the real page by setting its HTML `class` attribute to include the class `AdJailDefaultZone` and ensuring the element's policy grants `subtree` write access. If the real page has no valid and unique default zone, content written to the shadow page default zone will not appear on the real page.

### 4.3 Synchronization

After initial rendering of the real and shadow pages in the browser, the two pages are kept synchronized by exchanging the messages listed in Table 2. We conserve the total number of generated ad impressions, using an approach given in §4.3.1. Content written by ad script to the shadow page is mirrored to the real page by a process described in §4.3.2. User interface events are forwarded from the real page to the shadow page as detailed in §4.3.3. Lastly, §4.3.4 describes how content position and style is kept synchronized on both pages as needed by some ad scripts.

#### 4.3.1 DOM interposition

A primary goal of our approach is to conserve the number of ad impressions detected by an ad server, which we achieve using DOM interposition. Ad networks bill advertisers, and in turn pay publishers, based in part on the number of ad impressions. Impression counts are correlated to the number of requests for ad resources submitted to the web server [18]. When ad content is rendered on the real page, any external resources not available in the browser's cache will be requested, causing an impression. This may occur for several possible reasons out of our control, such as: the user disabled the cache, the ad net-

**(a) Real page to shadow page:**

`DispatchEvent(` *event* `)`
Dispatch *event* to shadow page.

`SetScrollPos(` *x*, *y* `)`
Scroll hidden shadow page to coordinates ( *x*, *y* ).

`SetStyle(` *syncId*, *properties* `)`
Set style of shadow page element identified by *syncId* as specified in *properties*.

**(b) Shadow page to real page:**

`Initialize(` *step* `)`
Initialize communication channel (two steps)

`InsertNode(` *syncId*, *index*, *model* `)`
Insert node described by *model* as child *index* of element identified by *syncId*.

`ModifyAttribute(` *syncId*, *name*, *value* `)`
Set attribute *name* to *value* on element identified by *syncid*.

`ModifyStyle(` *syncId*, *name*, *value*, *priority* `)`
Set CSS property *name* to *value* and *priority* on element identified by *syncId*.

`ModifyText(` *syncId*, *index*, *data* `)`
Set text content to *data* on *index* child of element identified by *syncId*.

`RemoveNode(` *syncId*, *index* `)`
Remove node *index* child node of element identified by *syncId*.

`ReplaceChildren(` *syncId*, *models* `)`
Replace child nodes of element identified by *syncId* with children described in *models*.

`WatchEvent(` *syncId*, *type*, *phase* `)`
Register a listener for event *type* and *phase* (bubble / capture) on element identified by *syncId*.

Table 2: Synchronization messages sent between real and shadow pages via DOM `postMessage()` API.

work instructed the browser (via `Cache-Control` HTTP headers) not to cache a resource, or per-origin cache partitioning [19] is in effect.

Impressions will be generated when the ad is rendered on the real page. Therefore, when ad content is initially rendered on the shadow page, we must prevent the browser from submitting HTTP requests for external resources, as that would cause superfluous impressions. Our implementation supports conserving impression counts for the following elements in our whitelist: `<img>`, `<iframe>` and `<object>` (Flash). Additionally we conserve impression counts for background image CSS properties in our whitelist: `background`, `background- image`, `list-style` and `list-style-image`.

To prevent ad impressions on the shadow page, we interpose on the common interfaces ad scripts use to create content. First, we hook DOM object prototype interfaces [25] to prevent ad scripts from setting URI attributes. For instance, we interpose on the `src` property of `HTMLImageElement` objects, and `getAttribute()` and `setAttribute()` DOM methods. We also hook other interfaces that access URI attributes, such as `document.write()`, `document.writeln()`, and *`element`*`.innerHTML`, to increase completeness and transparency of the interposition.

When ad script writes a URI attribute using one of these APIs, we substitute the real URI value with a placeholder value. For `write()`, `writeln()`, and `innerHTML`, this substitution requires a character search and replace in HTML source code. Our current implementation of this operation makes use of regular expression based textual transformation, which works well in practice, but may not be very precise under all circumstances. As the purpose of this substitution is to conserve ad impressions, a loss in precision here may affect compatibility with ads, but not security. If more precision is required, works on in-browser source-to-source HTML transformation [14, 34] can be leveraged, at the cost of additional overhead.

One exception we make to the above scheme is for `<script>` elements. Our interposition does not block the setting of `src` attributes for scripts, because our goal is to enable ad scripts to execute in the shadow page. Thus scripts are the only source of ad impressions from the shadow page. Since our policy enforcement mechanism prevents ad scripts in the real page, each script is created only once, thereby conserving the number of ad impressions.

#### 4.3.2 Content mirroring

We mirror ad content from the shadow page to the real page using a 5-step process: (1) monitoring the shadow page for modifications by the ad script, (2) modeling the detected modifications, (3) sending the model to the real page, (4) enforcing policies on the model, and (5) modifying the real page to reflect the model.

**1. Monitoring the shadow page for modifications** We monitor the shadow page for dynamic modifications using DOM interposition logic (introduced in §4.3.1). In addition to APIs that affect element attributes, we also hook APIs that modify the document, such as *`element`*`.appendChild()`. Whenever ad script attaches a new DOM node using `appendChild()`, our monitoring code is invoked before the actual modification takes place. Alternatively, *DOM mutation events* [51] can be leveraged to perform the same monitoring function with lower complexity than DOM interposition. However, Internet Explorer does not yet support mutation events, which would result in decreased compatibility.

**2. Modeling the detected modifications** When modifications to the shadow page are detected, we encode those changes using the same model format described in §4.2 and depicted in Figure 5. However, when we find content that was substituted by our interposition (ref. §4.3.1), we model the ad script's *intended* content instead of the substituted content. Models are passed to the real page, where the modifications will be reflected to the extent allowed by policies.

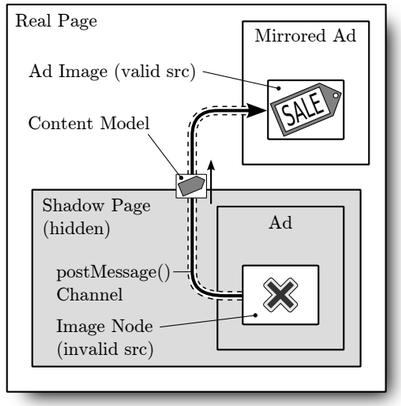



Figure 6: Rendering an ad image only on the real page so that just one impression is generated.

**3. Sending models to the real page** The process of sending a model of an image element is depicted in Figure 6. In the shadow page, we serialize the model data structure to a JSON string. We send the serialized model from the shadow page to the real page using the `InsertNode()` message from Table 2b. (Other types of modifications use the additional `postMessage()` notifications listed in Table 2b.) On the receiving end (i.e., the real page), we deserialize the string to recover the model data structure.

**4. Enforcing policies on the models** Our policy enforcement code in the real page receives the model from the shadow page. The model is then checked for any content that violates the real page policy annotations. We trim all policy-violating content from the model. For instance, if the model describes an image to be added to the page where the `enable-images` permission is denied, then we remove the image from the model. If the model describes an ad that is 1000 pixels wide and the policy only allows the ad to be 600 px, we allow the ad but restrict its maximum width to 600 px.

**5. Modifying the real page to reflect the modeled changes** Finally we merge the changes represented by the model into the real page. We create or modify constructs using DOM APIs, such as `document.createElement()` and *`element`*`.setAttribute()`. To ensure scripts are not injected into the real page during this process, we leverage the techniques we developed in BLUEPRINT [46] to enforce a *no-script* policy over all merged changes. This entails protecting several script injection vectors, including `<script>` elements, event handler attributes, `javascript:` URI schemes, CSS expressions, and more.

Mirroring ad content on the real page has the side-effect of modifying the real page script execution environment. For instance, elements such as `<input name="query" ...>` can pollute the namespace by creating properties such as `document.elements.query`. A straightforward solution to this problem is disallowing `name` and `id` attributes on mirrored ad content; however, this may reduce compatibility with some ads.

#### 4.3.3 Event forwarding

To prevent code injection attacks during content mirroring, our whitelist intentionally omits event handlers such as `onclick` and `onmouseover` that have been attached to ad content. In order to preserve event handler functionality in spite of this restriction, we perform event forwarding.

Event forwarding leverages our DOM interposition framework. We interpose on script operations used to register event handlers such as handler attributes and object properties (e.g., `onclick`, `onload`, etc.), using the same mechanism used for URI attributes and properties described in §4.3.1. Additionally, browser-specific APIs such as *`element`*`.addEventListener()` and *`element`*`.attachEvent()` are detected and interposed on when present.

When ad script uses any of these APIs to register an event handler on an element, and that element is also mirrored on the real page, we register our own handler for the same event on the mirrored element. Event handlers are registered on the real page when specified in content models (`InsertNode()` and `ReplaceChildren()` messages), or by sending the `WatchEvent()` message of Table 2. Whenever the event occurs on the real page, our handler is invoked and sends details of the event to the shadow page using the `DispatchEvent()` message (indicated by path ③ on Figure 3). On the shadow page we establish the appropriate JavaScript scope, then dispatch the event to the target element. This in turn invokes the ad script's original event handler. Effects caused by the ad script's handler are detected and mirrored back to the real page using the mechanism described in §4.3.2.

**Ad clicks** Unlike other user interface events, we do not forward click events on `<a>` (link) elements. Instead we click (i.e., activate) links on the real page, subject to enforcement of the `link-target` permission. This has the effect of bypassing any click event handlers the ad script may have registered on the activated link. Therefore there can be a compatibility trade-off in enforcing the

`link-target` permission if the ad script depends on such event handlers.

#### 4.3.4 Position and style synchronization

Some ads mimic the appearance of a pop-up window by temporarily overlaying parts of the web page. Although the pop-up window can appear at variable locations on the page, typically it is positioned such that it is visible (given the portion of the page that is scrolled into view) and relative to some other content (such as a contextual keyword). The ad script contains logic to compute the pop-up location based on the above criteria. However, if content appears at a different location on the real page than it does on the shadow page, the pop-up will be positioned incorrectly when mirrored. For this reason we support synchronizing the visual aspect of both real and shadow pages, even though the shadow page remains hidden.

First, we keep the window sizes of each page synchronized by setting the shadow page size to 100% of the real page size. Second, we sync the scroll position of both pages by registering an event handler for the real page's `onscroll` event. Whenever the event fires, we send a `SetScrollPos()` message to the shadow page. Our code running in the shadow page receives this message and adjusts the shadow page vertical and horizontal scroll offsets to match the real page.

Next we have to ensure content on the shadow page occupies the same location and extent as the corresponding content on the real page. For example, consider the inline text ad (Figure 1, #3), which highlights keywords and makes a pop-up appear near a keyword when the user's mouse hovers over it. The precise location of the keyword depends on many things, such as the absolute coordinates of the element containing the text, height and width of the container element, font size of the text, dimensions / layout of other content in the container, and more. We synchronize these details by sending the absolute position, size and *computed style* of each mirrored element to the shadow page via the `SetStyle()` message. On the shadow page we apply these properties to content elements, while keeping record that these are not "authentic" properties that should be synchronized back to the real page during any future content mirroring operations.

This strategy works very well in practice but is not perfect. For instance, there may be text in the real page that flows around an image. If the policy in effect for the text content allows read access, and the image is not readable, then the image will not appear on the shadow page and thus the text will not flow in the same way. To resolve issues due to the layout becoming out of sync, the publisher can either make the image readable or customize the shadow page to more accurately reflect the real page.

## 5 Evaluation

We evaluated AdJail to assess performance in three major areas. In §5.1 we investigate the compatibility of our architecture with six popular ad networks, each of which serve a variety of ads. The security of our approach is tested in §5.2. We then measure ad display latencies in §5.3. Although many ad networks exist which were not tested, we believe the relatively small sample we evaluated offer good insights into the compatibility and performance of AdJail.

### 5.1 Compatibility

To evaluate how well AdJail works with existing ad scripts, we tested it on six popular ad networks: Yahoo! Network, Google AdSense, Microsoft Media Network, Federated Media Publishing, AdBrite and Clicksor. The first four used banner ads, while the latter two employed more complicated inline text ads. Yahoo!, Google and Microsoft were three of the top ten ad networks in terms of U.S. market reach in April 2009. With a total audience size of 192.8 million, Yahoo! reached 86.6% of the market, Google reached 85.3%, and Microsoft reached 72.4% [3].

Federated Media, AdBrite and Clicksor rank lower in terms of U.S. market reach (e.g., AdBrite ranked #21 with a reach of 47.2%), but were chosen as they represent the small publisher market and demonstrate unique functionality. They are not as pervasive, therefore they are more likely to exhibit compatibility problems and less tested features. In our experiments we focused on the following observations: whether the ad functioned correctly, the minimum permissions required to support the ad, and whether click and impression counts were affected by our approach.

Our prototype AdJail implementation is a sufficient proof-of-concept to demonstrate the feasibility of our approach. The prototype is designed and tested to work on recent releases of the Chrome, Firefox, Internet Explorer, Opera and Safari web browsers. It does not yet have the level of refinement that would be present in a production system, which exposes some compatibility limitations we describe below.

**Correct functionality** To evaluate correct functionality we embedded ad scripts from each ad network in a series of AdJail test pages, then compared the user experience to the same ad scripts when used without sandboxing. The four banner ad scripts (Yahoo!, Google, Microsoft and Federated) all made use of the default ad zone feature. In this experiment we observed two main types of ad banner: animated image and Flash.

All of the banner ads rendered on the real page without any noticeable differences from rendering the ad without AdJail. Interacting with Flash ads via the mouse and clicking on banners worked exactly the same as the non-sandboxed ads. One minor issue we are aware of is that the contextual targeting approach used by Google AdSense does not work with our current implementation. This is because AdSense performs contextual targeting on the server, using an offline cached copy of the publisher's page. This limitation can be overcome by providing pre-computed shadow pages to ad networks who perform server-side contextual targeting, like AdSense.

For each of the inline text ad scripts (AdBrite and Clicksor), we annotated a news article with a full read and write access policy. The ad scripts identified keywords in the article and transformed them into interactive ads that "pop up" when the user hovers the mouse cursor over a keyword. This allowed us to evaluate the intricate synchronization capabilities of our architecture, such as ad script modifying existing page content and event forwarding. The pop-ups consisted of a decorative window border around the actual advertisement. AdBrite worked well in this experiment; its ads were simply `<iframe>`s wrapped by the decorative border. Clicksor also worked without any noticeable differences.

**Minimum permissions** For each tested ad network, we enabled the strictest set of permissions that would permit ads to function without impairment. These permissions are summarized in Table 3. To arrive at the set of permissions, we started with the base read and write access needed by the ad. We then enabled support in the content whitelist based on the needs of the ad. Finally, for fixed-size banner ads we set the maximum width and height policies.

Google AdSense was configured to serve text ads, so we were hoping to confine it with a strict text-only policy. Unfortunately the text ads were contained in an `<iframe>`, thus we had to set the `enable-iframe` permission.

AdBrite and Clicksor needed append write permission on the `<body>` element to create their pop-ups. Whitelist customization was also required for the pop-ups, as they contained custom HTML elements to prevent inheritance of publishers' CSS formatting rules [4]. AdBrite was easier to support as we only had to whitelist their custom `<ispan>` element. Clicksor used a randomly generated element tag name consisting of the word "span" followed by digits (e.g., `<span40110>`). To accommodate Clicksor we modified the whitelist to accept element tag names that matched the JavaScript regular expression `/^span[0-9]{5,7}$/`. Also we note that Clicksor was the only ad network to require `<form>` and `<input>` elements in its whitelist.

**Click and impression counts** To measure the number of clicks and impressions caused by ads, we configured our browser to route all traffic through a web proxy running the Squid proxy software. We rendered each ad script with and without sandboxing, and clicked on the displayed ads in each case. For this experiment, the web page hosting the ad script was completely blank except for a single paragraph of text, which was used for rendering inline text ads and contextual ad targeting.

A given ad script may show a different ad each time it is rendered. To ensure consistency in our evaluation, multiple renderings were sometimes performed for an ad network to ensure we clicked on the same advertisement with and without sandboxing. In between renderings, we cleared the browser's cache to ensure proxy access patterns were not affected by prior tests.

After performing the experiment, we analyzed the proxy's access logs. We discarded all log entries that referred back to our server hosting the test pages and AdJail source code. Comparing the remaining log entries, we did not find any differences in the HTTP requests generated by sandboxed versus non-sandboxed ads. Thus we conclude that in our experiment, ads using our sandbox environment did not impose any additional impressions or generate any additional clicks, thereby preserving traffic patterns crucial to the web advertising revenue model.

### 5.2 Security

To evaluate the security provided by AdJail we installed the RoundCube webmail v0.3.1 software on our web server. We integrated two ad network scripts on the main webmail interface: one ad script was included directly on the page, and the other was embedded using AdJail. A single trial consisted of replacing each of the two ad scripts with a malicious script designed to perform one specific attack or policy violation. We then observed if the malicious script functioned correctly in the non-sandboxed location, and whether the attack was prevented in the sandboxed location. Several trials were conducted to assess different attack vectors, and to determine the least restrictive policy required to defend each vector.

Our experiments were designed to support our claims in §1 of strong defense against several potent attack vectors to which ad publishers are routinely exposed. However, we did not evaluate the threats discussed in §2 that are beyond the scope of our current work: drive-by downloads, Flash exploits, privacy attacks, covert channels, and frame busting.

Results of the security evaluation are included on the right side in Table 3. With appropriate policies in effect, AdJail blocked all of the in-scope threats. We note that for each ad, write access was allowed for the subtree rooted at the `<div>` element designated for ad content. However, every ad policy denied write access (the default setting) for the rest of the document. A degree of leniency is required in our policies for compatibility with existing ads, which opens the door to some of the secondary attacks. However, every ad network we tested was protected from our primary threats: confidential data leaks and content integrity violations.

| Ad Network | Element | Computed Policy (Annotated policy in bold) | | | | | | | | Attack resistance | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *read access* | *write access* | *enable images* | *enable iframe* | *enable flash* | *max width* | *max height* | *over-flow* | E X | C B | I V | C J | U I | A P | O A |
| AdBrite | `<body>` | none | **append** | **allow** | **allow** | deny | none | none | deny | ✓ | ✓ | ✓ | | | | |
| | Article `<div>` | **subtree** | **subtree** | deny | **deny** | deny | none | none | deny | ✓ | ✓ | | | | | |
| Clicksor | `<body>` | none | **append** | **allow** | deny | deny | none | none | deny | ✓ | ✓ | ✓ | ✓ | | | |
| | Article `<div>` | **subtree** | **subtree** | deny | deny | deny | none | none | deny | ✓ | ✓ | | ✓ | | | |
| Federated Media | Ad `<div>` | none | **subtree** | **allow** | **allow** | **allow** | **90px** | **728px** | deny | ✓ | ✓ | ✓ | | | ✓ | ✓ |
| | Rest of page | none | none | deny | deny | deny | none | none | deny | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Google | Ad `<div>` | none | **subtree** | deny | **allow** | deny | **600px** | **160px** | deny | ✓ | ✓ | ✓ | | | ✓ | ✓ |
| | Rest of page | none | none | deny | deny | deny | none | none | deny | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Microsoft Media | Ad `<div>` | none | **subtree** | deny | **allow** | **allow** | **300px** | **250px** | deny | ✓ | ✓ | ✓ | | | ✓ | ✓ |
| | Rest of page | none | none | deny | deny | deny | none | none | deny | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Yahoo! | Ad `<div>` | none | **subtree** | deny | deny | **allow** | **90px** | **780px** | deny | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| | Rest of page | none | none | deny | deny | deny | none | none | deny | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 3: Policy annotations required to support several popular ad networks, and attacks prevented in policy enforcement regions. Attacks prevented are: EX: Execute arbitrary code in context of real page (non-XSS), CB: Data confidentiality breach, IV: Content integrity violation, CJ: Clickjacking, UI: UI spoofing, AP: Arbitrary ad position, OA: Oversized ad. Default `link-target` policy used for all.

Below we briefly describe our objectives and methodology for testing each attack.

**Execute arbitrary code in context of real page** In this attack we attempted to break out of the sandbox, by causing the browser to execute ad script code in context of the real page. This attack is critical because, if successful, malicious code can disable all policy enforcement logic in the real page and subsequently mount any of the other attacks. Specifically excluded from this vector is code injection by reflected, DOM0, and stored XSS attacks, which the web application can defend by other means.

We attempted to inject script code in the real page via DOM traversal, but this was blocked by the browser's SOP policy. Next, we evaluated 7 different real-world attacks sourced from the XSS Cheat Sheet [16]. Each attack demonstrated a unique code injection vector, such as embedded `script` element, event handler, `javascript:` URI, CSS expression, and more. These code injection attempts were blocked by enforcing a *no-script* policy on content models when constructing the mirrored ad in the real page, using the technique we developed in prior work [46].

To evaluate our defense against Flash-based script injection attacks, we created a Flash application that uses the `ExternalInterface` API to extract confidential data from the DOM. Flash regulates access to this API via the `allowScriptAccess` attribute of `<object>` elements, and `value` attribute of `<param>` elements when the `name` attribute is set to `allowScriptAccess`. Without ADJAIL, the ad network's script can create Flash objects on the real page with `allowScriptAccess` set to `always`. This setting permits Flash ActionScript code to fully access the real page's JavaScript environment, including sensitive page content via the DOM. Our defense blocks this attack vector by forcing the `allowScriptAccess` attribute to `never` on all `<object>` elements and relevant `<param>` elements. This action effectively disables the Flash `ExternalInterface` API.

All script injection attacks were prevented even with the most permissive policy that can be written using our policy language. Thus the script injection vector is defended for every possible policy configuration.

**Confidential information leak** For this attack we retrieved two items of confidential data from the real page: the user's session cookie and list of email contacts. Due to SOP restrictions, the sandboxed attack could not access the information by DOM traversal. (We note DOM traversal is also an ineffective strategy for all remaining evaluated attacks.) The only way the attack could access confidential data was when the data was given a policy granting full read access.



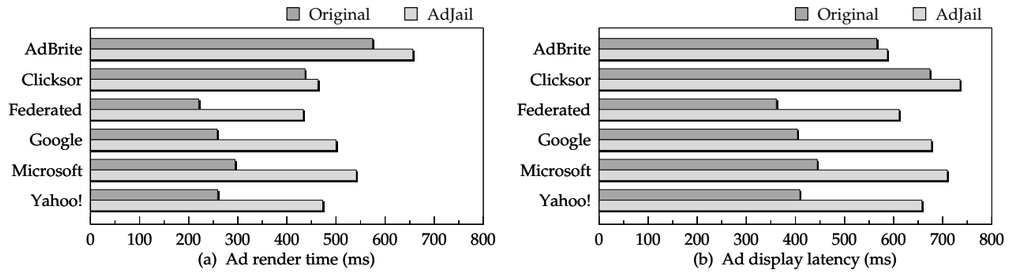



Figure 7: Rendering latencies: (a) time spent loading the ad, and (b) time from start of page load until ad appears.

**Content integrity violation** This attack tampers with trusted content on the real page: the user's email message headers. Specifically the attack makes all messages appear to be sent by prominent government officials. The sandboxed attack was unsuccessful except when the message headers were given a policy with full write access.

**Clickjacking** The clickjacking attack attempts to entice the user to unknowingly click on an `<iframe>` element. The attack script is based on detailed technical analysis of the vector [17, 54]. With a policy that disallows `<iframe>` elements, the sandboxed attack was unsuccessful because the policy prevents any `<iframe>` on the (hidden) shadow page from being brought up to the real page where the user can click it. Since any `<iframe>` embedded by the ad is unclickable to the end user, typical tricks to mask the clickjacking attack (e.g., hiding the `<iframe>` using transparency) are not a factor.

**User interface spoofing** We made an ad appear identical to trusted webmail user interface components in an attempt to lure users into interacting with the ad (i.e., an *interface spoofing* attack [26]). This attack was defeated by denying images, `<iframe>`s and Flash, and further constraining the ad with policies that disallow the ad from overlapping other parts of the trusted interface. Since the ad can still make use of textual elements, we note there exists a very small likelihood for an attacker to succeed through very nuanced UI spoofing attack using very small (single pixel) elements or text, such that images can be rendered in HTML one pixel at a time. Mitigating this threat may require advanced analysis of ad content or restricting the color palette available to ads.

**Arbitrary ad position** We made an ad appear on the real page outside of its write-accessible container element. This type of violation can be performed by setting an ad content display position that is outside the bounds of its container. With a policy that denies overflow, violations due to out-of-bounds display positioning are blocked. Position policies can also be violated by a *node splitting* attack, which may only succeed when there is no mechanism to provide hypertext markup isolation [41, 45]. Our content mirroring approach provides the necessary isolation by default to prevent node splitting attacks.

**Oversized ad** We made an ad larger than the publisher's expected ad size. The size violation was blocked by configuring a policy to limit the maximum height and width, and disallowing overflow.

### 5.3 Rendering overhead

To measure ad rendering latencies incurred by our policy enforcement mechanism, we placed each ad script on a typical blog page instrumented with benchmarking code. There were a total of 12 instances of the blog page: for each of the six ad networks evaluated in §5.1, one version of the blog page used the original ad, and a second version used ADJAIL to enforce the policies in Table 3. As the blog page is rendered, the ad script executes and scans for contextual data, requests a relevant ad from the ad network based on this data, and finally renders the ad. This experiment reflects the typical delays a end-user would experience when browsing publisher pages that integrate ADJAIL.

The test pages were rendered in Firefox v3.6.3 on an AMD Phenom X4 940 (3.0 GHz) workstation with 7.5 GB RAM. To resemble a typical browsing environment, the browser cache was enabled during the experiment. Each test page includes a link to our ADJAIL implementation source code (102 kB of JavaScript), which was cached by the web browser. The code is not optimized for space and contains much debug code. The memory overhead required by ADJAIL was reasonably consistent across ad networks, averaging 5.52% or roughly 3.06 MB.

Results of this experiment are shown in Figure 7. First we measured the time taken to render only the ad (Figure 7a). For AdBrite and Clicksor (inline text ads), this measurement consists of the time between the user triggering an ad pop-up and appearance of the pop-up. Although we do not separately report the latency incurred by forwarding events to the shadow page (ref. §4.3.3), this overhead is included in Figure 7. For this experiment, we stopped the benchmark after the ad's `<iframe>`

or `<object> onload` event was triggered, signaling the ad was complete. Without sandboxing, ads rendered in 374 ms on average. With ADJAIL, ad rendering averaged 532 ms, an additional latency of 158 ms.

To better understand the impact of ad rendering latency, we measured the time between when the page started loading until the ad completed rendering (Figure 7b). This is an important benchmark for ads, as many ad networks use a content distribution network (CDN) to improve performance in this regard [47]. For AdBrite, and Clicksor, we measured the time until inline text links finished rendering, although no ads are visible until the user triggers a pop-up. Without sandboxing, ads appear in 489 ms on average after the page begins to load. With ADJAIL, an additional 163 ms delay was incurred on average.

Optimizing performance is an important area for future work. A straightforward way to improve performance will be to optimize our prototype implementation. More significant gains may be achieved by adapting our approach to support pre-computing policies and shadow pages. It may be feasible to integrate caching of policies and shadow pages into web application templates and frameworks, to allow better performance without raising the publisher effort required to deploy ADJAIL.

## 6 Conclusion

In this paper, we presented ADJAIL, a solution for the problem of confinement of third-party advertisements to prevents attacks on confidentiality and integrity. A key benefit of ADJAIL is compatibility with the existing web usage models, requiring no changes to ad networks or browsers employed by end users. Our approach offers publishers a promising near term solution until web standards support for confinement of advertisements evolves to offer solutions agreeable to all parties.

## Acknowledgements

We thank Rohini Krishnamurthi for many insightful discussions that helped to shape principal ideas of this work. Our sincere thanks to our shepherd Lucas Ballard, and the anonymous reviewers for their helpful and thorough feedback on drafts. This work was partially supported by National Science Foundation grants CNS-0716584, CNS-0551660, CNS-0845894 and CNS-0917229. The first author was additionally supported in part by a fellowship from the Armed Forces Communications and Electronics Association.

### References

[1] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *17th USENIX Security Symposium*, San Jose, CA, USA, July 2008.

[2] Click Quality Team. How fictitious clicks occur in third-party click fraud audit reports. Technical report, Google, Inc., August 2006.

[3] comScore. April 2009 U.S. ranking of top 25 ad networks. `http://www.comscore.com/Press_Events/Press_Releases/2009/5/Top_25_US_Ad_Networks`, May 2009. Retrieved 19 Nov. 2009.

[4] Sean Conaty. Introducing the `<ispan>`. `http://nerdcereal.com/introducing-the-ispan/`, January 2008. Retrieved 1 Jun. 2010.

[5] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *19th International World Wide Web Conference*, Raleigh, NC, USA, April 2010.

[6] Douglas Crockford. ADsafe. `http://www.adsafe.org/`. Retrieved 1 Jun. 2010.

[7] Douglas Crockford. The application/json media type for JavaScript object notation (JSON). `http://tools.ietf.org/html/rfc4627`, July 2006. RFC 4627.

[8] Úlfar Erlingsson, V. Benjamin Livshits, and Yinglian Xie. End-to-end web application security. In *11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.

[9] Facebook Developers. Facebook JavaScript. `http://wiki.developers.facebook.com/index.php/FBJS`. Retrieved 8 Apr. 2010.

[10] Adrienne Felt, Pieter Hooimeijer, David Evans, and Westley Weimer. Talking to strangers without taking their candy: Isolating proxied content. In *1st International Workshop on Social Network Systems*, Glasgow, Scotland, April 2008.

[11] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *7th ACM Conference on Computer and Communications Security*, Athens, Greece, November 2000.

[12] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing capability leaks in secure JavaScript subsets. In *17th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, March 2010.

[13] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and detecting malicious Flash advertisements. In *25th Annual Computer Security Applications Conference*, Honolulu, HI, USA, December 2009.

[14] Google Caja. A source-to-source translator for securing JavaScript-based web content. `http://code.google.com/p/google-caja/`. Retrieved 1 Jun. 2010.

[15] Saikat Guha, Bin Cheng, Alexy Reznichenko, Hamed Haddadi, and Paul Francis. Privad: Rearchitecting online advertising for privacy. Technical Report MPI-SWS-2009-004, Max Planck Institute for Software Systems, Kaiserslautern-Saarbruecken, Germany, October 2009.

[16] Robert Hansen. XSS (cross site scripting) cheat sheet esp: for filter evasion. `http://ha.ckers.org/xss.html`, 2008. Retrieved 8 Apr. 2010.

[17] Robert Hansen and Jeremiah Grossman. Clickjacking. `http://www.sectheory.com/clickjacking.htm`, September 2008. Whitepaper.

[18] Interactive Advertising Bureau. Interactive audience measurement and advertising campaign reporting and audit guidelines. Global Version 6.0b, IAB, September 2004.

[19] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting browser state from web privacy attacks. In *15th International World Wide Web Conference*, Edinburgh, Scotland, May 2006.

[20] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for Web mashups. In *16th International World Wide Web Conference*, Banff, AB, Canada, May 2007.

[21] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *16th International World Wide Web Conference*, Banff, AB, Canada, May 2007.

[22] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. JavaScript instrumentation in practice. In *6th Asian Symposium on Programming Languages and Systems*, Bangalore, India, December 2008.

[23] Jeremy Kirk. Ad exploits Internet Explorer vulnerability to explose millions to adware. `http://www.infoworld.com/print/23520`, July 2006. Retrieved 23 Apr. 2010.

[24] Mary Landesman. ScanSafe: Weekend run of malvertisements. `http://blog.scansafe.com/journal/2009/9/24/weekend-run-of-malvertisements.html`, September 2009. Retrieved 23 Apr. 2010.

[25] Travis Leithead. Document Object Model prototypes, Part 1: Introduction. `http://msdn.microsoft.com/en-us/library/dd282900%28VS.85%29.aspx`, November 2008. Microsoft Corporation. Retrieved 22 May 2010.

[26] Elias Levy and Iván Arce. Interface illusions. *IEEE Security and Privacy*, 2:66–69, 2004.

[27] Zhenkai Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *19th Annual Computer Security Applications Conference*, Las Vegas, NV, USA, December 2003. IEEE Computer Society.

[28] V. Benjamin Livshits and Salvatore Guarnieri. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[29] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Language-based isolation of untrusted JavaScript. In *22nd IEEE Computer Security Foundations Symposium*, Port Jefferson, NY, USA, July 2009.

[30] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Run-time enforcement of secure JavaScript subsets. In *3rd Workshop in Web 2.0 Security and Privacy*, Oakland, CA, USA, May 2009.

[31] Dahlia Malkhi and Michael K. Reiter. Secure execution of Java applets using a remote playground. *IEEE Transactions on Software Engineering*, 26(12):1197–1209, December 2000.

[32] Gervase Markham. Content restrictions. `http://www.gerv.net/security/content-restrictions/`, March 2007.

[33] Leo A. Meyerovich and V. Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2010.

[34] Microsoft Live Labs. Web Sandbox. `http://websandbox.livelabs.com`. Retrieved 1 Jun. 2010.

[35] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In *ACM Symposium on Information, Computer and Communications Security*, Sydney, Australia, March 2009.

[36] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iFRAMEs point to us. In *17th USENIX Security Symposium*, San Jose, CA, USA, July 2008.

[37] C. Reis, J. Dunagan, Helen J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, USA, November 2006.

[38] Matthew Rogers. Facebook's response to uproar over ads. `http://endofweb.co.uk/2009/07/facebook_ads_2/`, July 2009. Retrieved 6 Apr. 2010.

[39] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: A study of clickjacking vulnerabilities on popular sites. In *4th Workshop in Web 2.0 Security and Privacy*, Oakland, CA, USA, May 2010.

[40] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *4th ACM Symposium on Operating Systems Principles*, Yorktown Heights, NY, USA, October 1973.

[41] Prateek Saxena, Dawn Song, and Yacin Nadji. Document structure integrity: A robust basis for cross-site scripting defense. In *16th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, February 2009.

[42] Barry Schnitt. Debunking rumors about advertising and photos. `http://blog.facebook.com/blog.php?post=110636457130`, November 2009. Retrieved 6 Apr. 2010.

[43] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the Web with content security policy. In *19th International World Wide Web Conference*, Raleigh, NC, USA, April 2010.

[44] Weiqing Sun, Zhenkai Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An efficient approach for realizing safe execution environments. In *12th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2005.

[45] Mike Ter Louw, Prithvi Bisht, and V. N. Venkatakrishnan. Analysis of hypertext isolation techniques for cross-site scripting prevention. In *2nd Workshop in Web 2.0 Security and Privacy*, Oakland, CA, USA, May 2008.

[46] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2009.

[47] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *17th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, March 2010.

[48] Ashlee Vance. Times Web ads show security breach. `http://www.nytimes.com/2009/09/15/technology/internet/15adco.html`, September 2009. NY Times. Retrieved 1 Jun. 2010.

[49] Yi-Min Wang, Doug Beck, Xuxian Jiang, and Roussi Roussev. Automated Web patrol with Strider HoneyMonkeys: Finding Web sites that exploit browser vulnerabilities. In *13th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2006.

[50] Wikipedia contributors. Same origin policy. `http://en.wikipedia.org/w/index.php?title=Same_origin_policy&oldid=190222964`, February 2008.

[51] World Wide Web Consortium. Document object model (DOM) level 2 events specification. `http://www.w3.org/TR/DOM-Level-2-Events`, November 2000.

[52] Yankee Group. Yankee Group forecasts US online advertising market to reach $50 billion by 2011. `http://www.yankeegroup.com/pressReleaseDetail.do?actionType=getDetailPressRelease&ID=1805`, January 2008. Retrieved 6 Apr. 2010.

[53] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *34th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, Nice, France, January 2007.

[54] Michal Zalewski. Browser security handbook. `http://code.google.com/p/browsersec/wiki/Main`, 2009. Retrieved 26 Jan. 2010.

# Realization of RF Distance Bounding

*Kasper Bonne Rasmussen*
*Department of Computer Science*
*ETH Zurich*
*8092 Zurich, Switzerland*
*kasperr@inf.ethz.ch*

*Srdjan Čapkun*
*Department of Computer Science*
*ETH Zurich*
*8092 Zurich, Switzerland*
*capkuns@inf.ethz.ch*

## Abstract

One of the main obstacles for the wider deployment of radio (RF) distance bounding is the lack of platforms that implement these protocols. We address this problem and we build a prototype system that demonstrates that radio distance bounding protocols can be implemented to match the strict processing that these protocols require. Our system implements a prover that is able to receive, process and transmit signals in less than $1ns$. The security guarantee that a distance bounding protocol built on top of this system therefore provides is that a malicious prover can, at most, pretend to be about $15cm$ closer to the verifier than it really is. To enable such fast processing at the prover, we use specially implemented concatenation as the prover's processing function and show how it can be integrated into a distance bounding protocol. Finally, we show that functions such as XOR and the comparison function, that were used in a number of previously proposed distance bounding protocols, are not best suited for the implementation of radio distance bounding.

## 1 Introduction

Distance bounding denotes a class of protocols in which one entity (the verifier) measures an upper-bound on its distance to another (untrusted) entity (the prover). In recent years, distance bounding protocols have been extensively studied: a number of protocols were proposed [3, 13, 10, 19, 30, 15, 25, 17, 12, 29] and analyzed [8, 26, 11, 23]. The use of distance bounding was suggested for secure localization [28], location verification [25], wormhole detection [16, 27], key establishment [22, 32] and access control [22].

Regardless of the type of distance bounding protocol, the distance bound is obtained from a rapid exchange of messages between the verifier and the prover. The verifier sends a challenge to the prover, to which the prover replies after some processing time. The verifier measures the round-trip time between sending its challenge and receiving the reply from the prover, subtracts the prover's processing time and, based on the remaining time, computes the distance bound between the devices. The verifier's challenges are unpredictable to the prover and the prover's replies are computed as a function of these challenges. In most distance bounding protocols, a prover XORs the received challenge with a locally stored value [3] or uses the received challenge to determine which of the locally stored values it will return [13, 29]. Thus, the prover cannot reply to the verifier sooner than it receives the challenge, it can only delay its reply. The prover, therefore, cannot pretend to be closer to the verifier than it really is; only further away.

One of the main assumptions on which the security of distance bounding protocols relies is that the time that the prover spends in processing the verifier's challenge is negligible compared to the propagation time of the signal between the prover and the verifier. If the verifier overestimates the prover's processing time (i.e., the prover is able to process signals in a shorter time than expected), the prover will be able to pretend to be closer to the verifier. If the verifier underestimates this time (i.e., the prover needs more time to process the signals than expected), the computed distance bounds will be too large to be useful.

The challenge in implementing distance bounding protocols is therefore to implement a prover that is able to receive, process and transmit signals in negligible time. This requirement can be easily met with ultrasonic distance bounding implementations where the prover's processing needs to be in the order of $\mu s$. However, because ultrasonic distance bounding is vulnerable to RF wormhole attacks [16, 27],

its application is limited to few specific applications (e.g., [22]). For most applications, radio distance bounding is the main viable way of verifying proximity to or a location of a device. In this case, the prover's processing time needs to be about $1ns$ which would, in the worse case, allow a malicious prover to pretend to be closer to the verifier by approx. $15cm$ (assuming that the malicious prover is able to process signals instantaneously). Currently available platforms do not support such fast processing. This strict processing requirement has been, so far, one of the main obstacles for the wider deployment of RF distance bounding protocols and related solutions.

In this work, we address this problem. We make the following contributions. We build a prototype system that demonstrates that radio (RF) distance bounding protocols can be implemented to match the prover's strict processing requirements (i.e., that the prover's processing time is below $1ns$). We use concatenation as the prover's processing function and implement it using a scheme that we call Challenge Reflection with Channel Selection (CRCS). Our implementation eliminates the need for signal conversion and demodulation since it does not require that the received challenges are interpreted by the prover before the prover responds to them. Our prover is therefore able to receive, process and transmit signals in less than $1ns$. We design a distance bounding protocol that uses concatenation, implemented with CRCS, as the prover's processing function and we analyze its security; we base this protocol on Brands and Chaum's original distance bounding protocol [3].

We further show that processing functions such as XOR and the comparison function, that were used in a number of proposed distance bounding protocols, are not best suited for the implementation of radio distance bounding. The main reason is that, although XOR and comparison can be executed fast, these functions require that the radio signal that carries the verifier's challenge is demodulated, which, with today's state-of-the-art hardware, results in long processing times (typically $\geq 50ns$). The design and implementation of the distance bounding protocol based on concatenation shows that the use of functions which require that the prover demodulates (interprets) the verifier's challenge before responding to it is not necessary for the implementation of distance bounding.

To our knowledge this work is the first to propose a realizable distance bounding protocol using radio communication, with a processing time at the prover that is low enough to provide a useful distance granularity.

The rest of the paper is organized as follows. In Section 2 we describe the basic operation of distance bounding protocols. In Section 3, we discuss prover's processing functions and their appropriateness for the implementation of radio distance bounding. In Section 4 we describe the design of our distance bounding protocol and in Section 5 we analyze its security. In Section 6 we present our implementation and our measurement results. In Section 7 we discuss related work and we conclude in Section 8.

## 2 Background on Distance Bounding Protocols

Distance bounding protocols were first introduced by Brands and Chaum [3] for the prevention of mafia-fraud attacks on Automatic Teller Machines (ATMs). The purpose of Brands and Chaum's distance bounding protocol was to enable the user's smart-card (verifier) to check its proximity to the legitimate ATM machine (prover).

The core of all distance bounding protocols is the distance measurement phase (shown in Figure 1), wherein the verifier measures the round-trip time between sending its challenge and receiving the reply from the prover. More precisely, the verifier challenges the prover with a $b$-bit freshly generated nonce $N_v$ (typically $b = 1$). Upon reception of the challenge, the prover computes a response $f^P(N_v)$, and sends it to the verifier. This process is repeated $k$ times. After the challenge-response exchange the verifier verifies the authenticity of the replies (in this step distance bounding protocols differ) and measures the time $t_s^V - t_r^V$ between the challenge and the response. Based on the measured times, the verifier estimates the upper-bound on the distance to the prover. The time $t_s^P - t_r^P$ between the reception of the challenge and the transmission of the response at the prover is either negligible compared to the propagation time $t_r^P - t_s^V$ or is lower bounded by the prover's processing and communication capabilities $\delta$, i.e., $t_s^P - t_r^P \geq \delta$.

After the execution of a distance bounding protocol the verifier knows that the prover is within a certain distance, namely:

$$dist = \frac{t_s^V - t_r^V - \delta}{2} \cdot c$$

where $\delta$ is the processing time of the prover (ideally 0) and $c$ is the propagation of the radio signal.

Although the designs of distance bounding protocols differ [3, 13, 10, 19, 30, 15, 25, 17, 12, 29], given their common distance measurement phase,



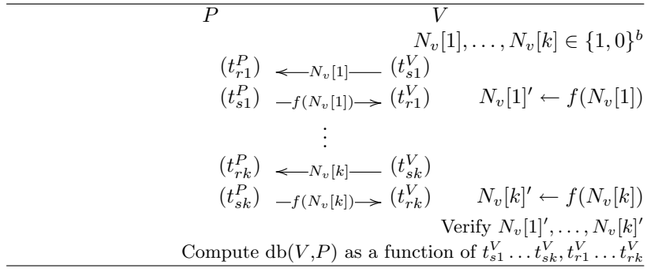



Figure 1: The distance measurement phase of distance bounding protocols consists of a rapid exchange of messages where the verifier measures the round-trip time between sending its challenges and receiving the replies from the prover.

their security relies on the same underlying ideas. We briefly summarize them here. Distance fraud attacks [3], in which the prover tries to pretend to be closer to the verifier, are prevented by the following: (i) the prover cannot generate the reply before it receives the challenge and (ii) the duration of time the verifier accounts that the prover will process the reply is not longer than the prover's actual processing time. The Mafia-fraud (or man-in-the-middle - MITM) attack [9], by which an attacker convinces the verifier that the prover is closer than it really is, is prevented since the attacker cannot predict exchanged challenges/replies and since it cannot speed-up the propagation of messages (the messages propagate at the speed of light over a radio channel). Given this, the attacker cannot shorten the distance measured between the verifier and the prover.

Distance bounding protocols therefore provide the verifier with an upper-bound on its physical distance to the prover.

## 3 Functions Appropriate for Distance Bounding Realization

As discussed in Section 2, one of the main assumptions on which the security of distance bounding protocols relies is that the time that the prover is allowed to spend in processing the verifier's challenge is negligible compared to the propagation time $t_r^P - t_s^V$ of the signal between the prover and the verifier. In most applications, the prover's processing time would therefore need to be around $1ns$. This would, in the worse case, allow a malicious prover to pretend to be closer to the verifier by approx. $15cm$ (assuming that the malicious prover is able to process signals instantaneously). Such short processing time cannot be achieved with existing platforms.

The main challenge is therefore to design distance bounding protocols which use prover processing functions $f(N_v)$ that can implemented such that they can be executed in $\leq 1ns$. Before presenting a function that is well suited for this purpose, we first discuss functions that were used in distance bounding protocols that are proposed in the open literature.

The first (obvious) candidate processing functions are various encryption functions, hash functions, message authentication codes and digital signatures; the use of digital signatures for this purpose was proposed by Beth and Desmedt in [1]. The use of such functions would largely simplify the design of distance bounding protocols; it would be sufficient to use well studied challenge-response authentication protocols [2] where the verifier would measure the round-trip time between the issued challenge and the received response. However, the processing time for these functions even with the fastest available implementations by far exceeds the required processing time.

In [3] Brands and Chaum proposed a distance bounding protocol that uses XOR as a processing function. In this protocol the prover XORs the verifiers challenge with the value that the prover wants to transmit back and sends the result back to the verifier. The main reasoning behind this choice was that XOR is a fast operation and that it should be feasible to execute it within the required processing time. Hancke and Kuhn [13] propose a distance bounding protocol where the prover, based on the verifier's challenge chooses from which of the two local registers it should send a value back. Again, one of the main reasons for choosing this function was that such a function (comparison and access) can be executed fast.

Although XOR and comparison can be executed fast, these functions require that the radio signal that carries the verifier's challenge is converted from an analog to a digital signal (ADC) and demodulated. Only when it is demodulated, the challenge can be used by the prover in an XOR function or for the selection of the register. Equally, in order to communicate the reply back to the verifier, the prover needs to modulate the signal and convert it from the digital to the analog signal (DAC). These steps, signal detection, ADC/DAC conversion and signal modulation/demodulation, increase the provers processing delay by approx. 170ns [24], not including possible RX/TX switching costs[1]. The implementations of an XOR or of a comparison function that require the signals to be digitalized and demodulated therefore require such processing which, using today's state-of-the-art hardware, is not sufficiently fast to meet the security requirements of distance bounding protocols. Even if some processing steps can be sped-up or removed, the prover will still need a way of (reliably) detecting if it received a challenge that corresponds to a bit "0" or a bit "1", which requires some processing and thus reduces the security guarantees of the protocol. Namely, every nanosecond of additional processing in the implementation of the prover means that a malicious prover with a faster implementation shorten the measured distance even further.

In what follows, we show that the choice of a concatenation function as the prover's processing function, when implemented using a scheme that we call *Challenge Reflection with Channel Selection (CRCS)* eliminates the need for signal conversion and demodulation since it does not require that the received challenges are interpreted by the prover before the prover responds to them. The prover, implemented using CRCS is therefore able to receive, process and transmit signals in less than $1ns$.

### 3.1 Prover: Concatenation Implemented using Challenge Reflection With Channel Selection

In this section we describe our implementation of concatenation as the prover's processing function.

Bit concatenation $CAT : N_p[i] \times N_v[i] \rightarrow r[i] = N_v[i]||N_p[i]$ takes as input the verifier's challenge bit $N_v[i]$ and the prover's input bit $N_p[i]$ and returns a two-bit reply $r[i] = N_v[i]||N_p[i]$. CAT is therefore given by the following table.

$CAT$ :

| $N_p[i] \backslash N_v[i]$ | 0 | 1 |
|---|---|---|
| 0 | 00 | 10 |
| 1 | 01 | 11 |

[1]We are not aware of the radio design that can perform these operations faster.

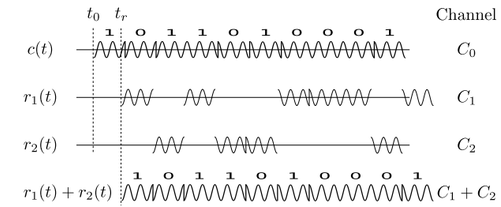



Figure 2: The verifier measures the time between sending a challenge signal $c(t)$ and receiving the reply signal $r(t) = r_1(t) + r_2(t)$. If $c(t) = r(t)$, the distance bound to the prover is then given by $(t_r - t_0) \cdot c$, where $c$ is the speed of light.

### 3.2 Verifier: Calculation of the Distance Bound

In order for concatenation to be useful for distance bounding, we implement it by Challenge Reflection with Channel Selection. Our implementation uses three (non-overlapping) communication channels. The verifier sends its challenge bits to the prover using one communication channel ($C_0$), whereas the prover replies using two communication channels ($C_1, C_2$) (Figure 2). While it is receiving the verifier's challenge bit (i.e., the signal that encodes it), the prover is responding with the same signal (bit), but it is sending it on either channel $C_1$ or channel $C_2$, depending on its current input bit $N_p[i]$. For every challenge bit that it received from the verifier, the prover therefore transmits two bits of the reply back to the verifier, encoded in the form of the signal (it reflect back the same signal that it received) and of the response channel (it chose the channel on which to reply). The response $r = 10$ is then interpreted as: the challenge bit 1 is reflected on channel $C_1$, where the channel $C_1$ denotes bit 0, and channel $C_2$ denotes bit 1). The prover therefore implements challenge reflection with channel selection. Note that, although the prover replies with two bits for each challenge bit, the duration of transmission of those two bits is the same as for a single bit of the verifier's challenge, since the second bit of the prover's reply is encoded in the form of channel selection. This is illustrated on Figure 2.

The schematic of our prover implementing CRCS is shown on Figure 3. The figure shows the signal in the frequency domain as it passes through various stages of the prover's circuit. The prover receives the challenge-signal (centered at the frequency $f_c$) on the receiving antenna. The received signal is then multiplied by $f_\Delta$ which creates two signals on two channels each with central frequencies $f_c + f_\Delta$ and $f_c - f_\Delta$, respectively. The current bit of the prover's nonce $N_p[i]$ determines which of the two channels are used to send the response signal on the transmitting antenna. The verifier's signal is thus reflected back on the channel selected by the prover. Here, the verifier's challenge bit can be encoded in the challenge signal using e.g., Pulse Amplitude Modulation (PAM) or Binary Phase Shift Keying Modulation (both of which are used with Ultra-Wide-Band ranging systems). The prover's response carries two bits, one encoded in the signal that it sends back (the same bit that it received by the verifier), and the other encoded in the channel on which it responds (i.e., $N_p[i]$).

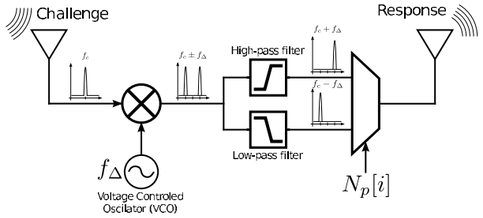



Figure 3: Schematic of the prover (i.e., of the implementation of concatenation as its processing function using CRCS). The figure shows the signal in the frequency domain at various stages of the circuit. The challenge-signal (with center frequency $f_c$) is received by the receiving antenna (on the left) and multiplied by $f_\Delta$. This multiplication shifts the signal by $\pm f_\Delta$ to the channels on two sides of the original channel. The bit of the prover's nonce $N_p[i]$ determines which of the two channels is used to send the response on the transmitting antenna (on the right).

Here, signal multiplication and selection are done using analog components only. Namely, the challenge signal passes through an analog mixer where it is multiplied with a local oscillator signal with a frequency $f_\Delta$. This mixer outputs two signals on frequencies $f_c + f_\Delta$ and $f_c - f_\Delta$, which are separated by a high-pass and a low-pass filter, respectively. Finally, the $N_p[i]$ bit (which the prover have committed to), determines which of the two signals will be transmitted back to the verifier.

Figure 2 shows the calculation of the distance bound by the verifier (the signals are shown in the time domain). The verifier notes the exact time $t_0$ when it starts transmitting the challenge bits $N_v[i], ...N_v[k]$ encoded in the signal $r_1(t)$, and then listens on the two reply channels $C_1$ and $C_2$ (that correspond to the frequencies $f_c + f_\Delta$ and $f_c - f_\Delta$). When a reply comes back (e.g., on channel $C_1$) the verifier will mark the exact time $t_r$ of the arrival of the signal. The verifier will then wait for the arrival of the entire challenge, noting for every time slot on which channel the reply was sent. After the entire nonce has been received and processed by the radio, the verifier checks that the data bits in the reply are the same as those sent in the challenge, i.e., that $c(t) = r_1(t) + r_2(t)$. If that is the case, the distance bound is then computed as $(t_r - t_0) \cdot c$, where $c$ is the speed of light. This bit comparison is important for the security of our distance bounding protocol (as we detail in Section 4); it can be efficiently done using autocorrelation, which can then simultaneously be used to calculate the time difference (e.g., as it is used in GPS [20]).

## 4 Distance Bounding Realization

In this section we present our distance bounding protocol and its realization. The protocol uses concatenation implemented using CRCS as the prover's processing function. The main security properties that we want our protocol to achieve are resilience to distance fraud and Mafia fraud attacks.

Our protocol is shown in Figure 4. It closely resembles the original protocol of Brands and Chaum [3], except that it does not use rapid bit exchange, but instead uses full duplex communication with signal streams. XOR is replaced with the concatenation function, and additional checks by the prover and the verifier are added to make sure the implementation of concatenation using CRCS does not introduce vulnerabilities.

The prover starts the protocol by picking a fresh (large) nonce $N_p$. The prover then sends a commitment to the nonce (e.g., a signed hash of the nonce) to the verifier. Already now, the prover will activate its distance bounding hardware and set the output channel according to the opposite of the first bit of the nonce $N_p$. From this moment, any signal that the prover receives on channel $C_0$ will be reflected on the output channel that is set. However, the prover does not yet start switching between output channels.

Upon receiving the commitment, the verifier picks a fresh (large) nonce $N_v$ and prepares to initiate the distance bounding phase in which it will measure

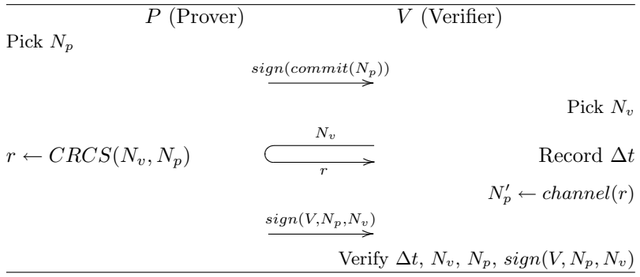



Figure 4: RF distance bounding protocol.

the distance bound to the prover. The verifier starts a high precision clock to measure the (roundtrip) time of flight of the signal and begins to transmit his nonce $N_v$ on channel $C_0$. From this point on, the verifier will also listen on the two reply channels $C_1$ and $C_2$ and will keep listening on the two channels until he either receives the expected response from the prover or until he detects an error and aborts the protocol.

As soon as the prover receives (and demodulates) the first bit of $N_v$ on $C_0$, he starts switching reply channels according to the bits of his nonce $N_p$. Here, we note that while the first few bits are being demodulated, the prover is still reflecting the input (challenge) bits, but he did not start the switching of the channels (i.e., he did not start sending back $N_p$). The demodulation of the bits is not done within the distance bounding hardware (that we call the distance bounding extension), but is done in the prover's regular radio. It is not important how long it takes for the prover's radio to demodulate the first bits, since the prover does not need to begin to switch the output channels within any predefined time (as long as the switching starts within the duration of $N_v$ and allows the transmission of $N_p$). Equally, the first part of $N_v$ could be known and constitute a public, fixed-length preamble upon the detection of which the prover would start switching the channels (i.e., would start sending $N_p$).

When the prover starts sending $N_p$, he will send the bits of $N_p$ with a fixed frequency (e.g., every $500ms$) by switching channels depending on the value of the current bit (Figure 2). In each interval, the prover will therefore reflect back several bits of $N_v$ and a single bit of $N_p$. The bit of $N_p$ is encoded in the choice of the reply channel. The prover will, in parallel, also receive the challenge on channel $C_0$ using his regular radio and will demodulate it.

When the verifier has sent all the bits of his nonce, he waits for the prover to complete the reflection of the signal and then both the prover and verifier disable their distance bounding extensions. The verifier can then use an auto-correlation detector like the ones used in GPS receivers [20] to determine the exact time of flight of the reflected signal. This can also be done during the distance bounding phase, i.e., in parallel to the analog distance bounding circuit.

After the (time-critical) distance bounding phase is complete the prover sends a signed message containing his nonce $N_p$, the identity of the verifier $V$ and the verifier's nonce $N_v$ to the verifier. The verifier must then check five things:

- That all the bits of $N_p$ reflected by the prover are of the same width (time duration). This is necessary to prevent mafia fraud and is described in more detail in Section 5.3.
- The data that was reflected back from the prover must be *exactly* the same as what was sent. I.e., when the signal $r(t) = r_1(t) + r_2(t)$ is demodulated, the message must contain $N_v$. This is visualized in Figure 2.
- The value of $N'_p$ obtained during the distance bounding phase must match the commitment sent in the first protocol message.
- The signature of the final message must be valid and it must correspond to the expected identity of the prover.
- The time of flight of the signal $\Delta t$ must be less than some predefined upper limit $t_{max}$. The upper limit is application dependent. E.g., it can be the radius of some region of interest, or it can be the (estimated) maximum transmission range of the radio.

The order is which these checks are performed is not important but all checks must pass for the distance bound to be accepted. If all the checks pass, the verifier calculates the distance to the prover as

$$d = \frac{\Delta t - \delta_p}{2} \cdot c \tag{1}$$

Where $c$ is the speed of light and $\delta_p$ is the very small processing delay of the prover. In our implementation $\delta_p < 1ns$ resulting in a maximum error on about $15cm$.

# 5 Security Analysis

In this section we analyze the resistance of our protocol to distance fraud and mafia fraud, as well as attacks against CRCS.

## 5.1 System And Attacker Model

We consider three nodes, the prover $P$, the verifier $V$ and the attacker $M$. The goals for the three participants are as follows: the verifier wants to acquire an upper bound on the distance to the prover, i.e., the verifier wants to know that the prover is closer than a certain distance. The prover wants to prove to the verifier that he is within a certain distance. The goal of the attacker is to disrupt this process such that the verifier obtains an incorrect distance bound. The verifier holds an authentic public key of the prover. The attacker and the prover do not collude. The attacker corresponds to the standard Dolev-Yao attacker that controls the network and thus can eavesdrop on all the communication between the prover and the verifier, can arbitrary insert and remove messages to/from the communication channel. She is equally free to transmit nonsensical signals. The attacker knows the public parameters of the distance bounding protocol and the type of hardware used by the nodes and thus the processing times of the prover's and verifier's radios. She is only limited by the fact that it does not have access to the secrets that are held by the prover and the verifier and cannot break cryptographic primitives.

We consider two attacks: Distance fraud, where the prover tries to shorten the measured distance bound, and Mafia fraud where the attacker tries to shorten the bound (but does not collude with the prover). We show that our protocol resists both attacks. There is a third type of attack in which the attacker colludes with the prover and has access to some, but not all, of the secret key material of the prover (e.g., only nonces and short-term secrets). This attack is often called the terrorist attack. We do not specifically address terrorist attacks, but it has been shown [4] that if needed, distance bounding protocols can be extended to generally protect against this attack.

## 5.2 Distance Fraud

Distance fraud is an attack performed by a malicious prover and consists of the prover trying to shorten the distance measured by the verifier.

The verifier uses equation (1) to calculate the distance to the prover. For the prover to "shorten" the distance to the verifier (without actually moving closer) he must manipulate the verifiers calculation and the only thing the prover can influence is $\Delta t$. For the prover to reduce the $\Delta t$ measured by the verifier, thereby reducing the distance, he must make his replies arrive at the verifier sooner than they otherwise would, i.e., he must guess the correct reply (i.e., guess the challenge) and send it before the verifier expects. In our protocol, the reply which the prover must send back is the signal he receives on channel $C_0$. In order to do this, the prover must guess the content of the challenge signal since the content of the reply is checked by the verifier as a part of the verification process. The content of the challenge is $N_v$ and the probability of successfully guessing that is given by $\frac{1}{2^{|N_v|}}$.

Attacks that rely on manipulation of the modulation scheme, e.g., "late commit" attacks described by Hancke and Kuhn [14] will not work on this protocol because the verifier uses auto-correlation to find the exact time-of-flight of the signal (as it is done in GPS receivers [20]) rather than using a peak or energy detector. This means that any manipulation done to, say, the first symbol of the response will not have any effect unless all subsequent symbols are also shifted forward. This would require the malicious prover to guess all the symbols in advance and can therefore only be done with negligible probability of $\frac{1}{2^{|N_v|}}$.

The same argument applies to attacks where the prover tries to guess the first bit of the nonce [8]. Because the prover doesn't store and forward the nonce, but instead must reflect it directly, the prover would have to guess all the bits of the verifier's nonce to perform the attack. We can therefore conclude that the prover can commit distance fraud only with probability $\frac{1}{2^{|N_v|}}$.

## 5.3 Mafia Fraud

Mafia fraud is an attack performed by an external attacker that physically resides closer to the verifier than the prover. The attack aims to make one of the parties (either the prover or the verifier or both) believe that the protocol was successfully executed

when, in fact, the attacker shortened the distance measurement. The requirement that the attacker be closer to the verifier than the prover is only necessary because, if the attacker is further away the attack is trivially defeated by the protection against distance fraud attacks.

In order for an external attacker to shorten the distance measured by the verifier, the attacker must respond before the prover during the distance bounding phase. However, because of the checks performed by the verifier at the end of (or during) the distance bounding phase, it is not sufficient to just reply before the prover, the attacker must also make the value of his nonce match the commitment sent by the prover in the beginning of the protocol. Since the attacker can not find a nonce to match the commitment sent by the prover, e.g., find a collision for the hash function used to generate the commitment, the attacker is forced to replace the provers commitment with his own, thereby passing the commitment check. However, the attacker cannot fake the prover's signature in the final message so he cannot confirm the nonce.

The attacker can get the prover to reply before the prover receives $N_v$, e.g., by sending his own early signal to the prover, however, this will result in the prover getting $N'_v \neq N_v$ which will be detected by the verifier in the final message. This assumes that any malicious change to the signal will result in a change in the demodulated nonce $N_v$. If that cannot be guarantied, e.g., because of the sample rate at the prover or the modulation scheme used for communication, the prover can record the raw incoming signal and send it back to the verifier. The verifier can then, e.g., use autocorrelation to make sure the signal received by the prover is the same as what the verifier sent.

We can therefore conclude that an attacker can only commit mafia fraud if he can break, either the commitment scheme or the signature scheme used in the protocol.

Because of the way the distance bounding radio extension is designed it is possible for an attacker to get the current bit of the provers nonce. As explained in Section 3.1, the prover's radio extension will shift any signal that arrives on the center channel to either channel $C_1$ or channel $C_2$ depending on the current bit of the provers nonce. An attacker can exploit this to get the current bit of the prover's nonce without the prover's knowledge. If the attacker sends a very weak signal, e.g., a DSSS [21] signal with a spreading code known only to the attacker, the attacker can determine what channel the response is sent back on, and therefore the current

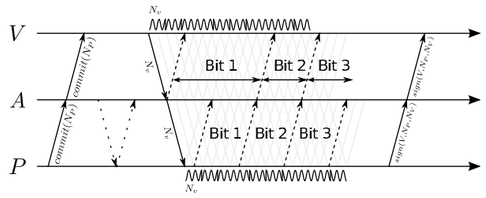



Figure 5: Man in the middle attack (Mafia fraud). The figure shows the timing of the messages sent by the verifier (V), the attacker (M) and the prover (P). Even if the attacker is able to learn the value of the first bit on the prover's nonce, the attack will fail because the attacker is forced to make the first bit longer than the subsequent bits if he wants to reply early.

bit of the prover's nonce. Unless this is prevented, the attacker can use this information to perform a successful mafia fraud attack.

In order to prevent this attack the prover must make sure not to expose all the bits of his nonce before they are needed. There are two ways this can be ensured: Either the prover must only enable his distance bounding hardware once he is sure that the verifier has started his transmission or he must make sure that his reply bits (of $N_p$) are of exactly the same duration. Of course the time duration must also be known and later checked by the verifier. Our protocol uses the second method. Figure 5 illustrates how this measure prevents the attack. In the example of this figure the attacker obtains the value of the first bit of the provers nonce, and uses it to reply early to the verifier's challenge. However, because the prover doesn't expose the second bit of his nonce until after the duration of the first bit has expired, the attacker is forced to make the first bit 'too long', thus getting detected.

In order to perform this attack, the attacker would need to guess all the bits of $N_p$, which she can do only with the probability $\frac{1}{2^{|N_p|}}$.

## 6 Implementation and Measurements

In this section, we describe our implementation of the prover and the related measurement results.

Our prototype can be seen on Figure 6. The central part of the prototype is the mixer (1) which is responsible for shifting the received challenge up and down in frequency. The signal from the receiving antenna comes in from the right (A) and passes



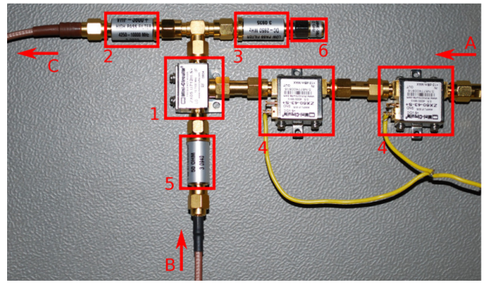



Figure 6: This picture shows the prototype implementation of the prover. It consists of a mixer (1), a high-pass filter (2), a low-pass filter (3), four amplifiers (4) (only two visible), a 1dB attenuator (5) and a terminating resistor (6). The signal from the receiving antenna (A) is mixed with the local oscillator (B) and sent to the transmitting antenna (C). The yellow wires are power (+5V). This prototype is an implementation of the scheme described in Figure 3.

through four amplifiers (4) to bring it up to a power level where is can be mixed by our mixer. The local 500MHz sine wave used for the mixing, comes in from the bottom of the figure (B) and is passed through a 1dB attenuator (5) to bring it to the same level as the radio signal before mixing. The output of the mixer is split in two and each is passed through either a high-pass filter (2) or a low-pass filter (3) to eliminate the unwanted channel. In this prototype we did not implement the switching mechanism. Instead channel $C_2$ is fed directly to the transmission antenna (C). In order for the signal to split properly, both sides must have a similar load. for this reason we added a 50Ω resistor (6) to terminate the unused channel $C_1$. The implementation of the switching mechanism can be done using a simple transistor based switch. We note, that the switch can only marginally increase the processing delay since, once set to a particular channel, the switch essentially acts as a piece of very short wire connecting the setup to the antenna. This prototype is an implementation of the scheme described in Figure 3.

### 6.1 Delay At The Prover

We first wanted to see if our prototype implementation could receive a signal, shift it to another channel and transmit it back to the verifier in $\leq 1ns$.

In order to test this, we first transmit the challenge and response signals through cables so as to better be able to control signal strength and reduce

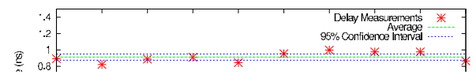
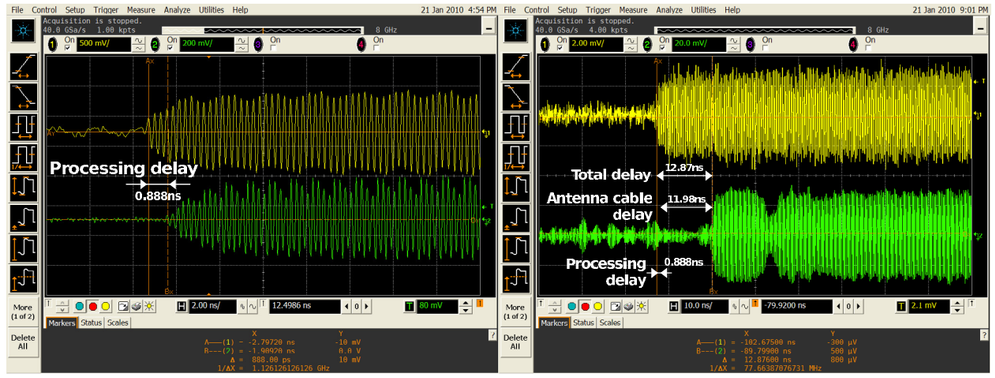



Figure 8: Processing time at the prover. The ten different delay measurements where done using our measurement setup described in Section 6.1. The figure shows that the variation in processing time is small ($\sigma = 61.22ps$) and that the average processing delay is $\mu = 912.92ps$. I.e., less than $1ns$.

noise (later we show that the same setup works using wireless communication as well). The challenge signal sent on channel $C_0$ is a 3.5GHz sine, modulated by a 1Hz pulse so it is easy to see and capture the start of a new "bit". Our response signal is sent back on channel $C_2$ at 4.0GHz (i.e., $f_c = 3.5GHz$ and $f_\Delta = 0.5GHz$). We generated the 3.5GHz challenge using a function generator. The generated signal is split by a power splitter and one end is fed, via a 1 meter cable, into our prototype. The other end was connected to a 40Gs/s oscilloscope, via another 1 meter cable, to provide the ground truth signal to which we compare the delay of our prototype. Because both cables have the same length, the 3.5GHz signal (the challenge) will arrive at the same time at the oscilloscope and at the reception point of our prototype. The output (the response) from the prototype is plugged directly into another input of the same oscilloscope (keeping the signal path as short as we could make it using this setup).

Figure 7(a) shows the two signals. The top (yellow) signal is coming directly from the function generator. It is an exact copy of the signal that arrives at the input of our prototype (this signal arrives at the oscilloscope and at the prototype input at the same time). The bottom (green) signal is what comes out of our prototype implementation. It is a 4.0GHz signal, i.e., the original signal shifted up by 500MHz. We see that the difference in arrival times between these two signals (i.e., the processing time of the prover) is $0.888ns$. As described in Section 2 the delay at the prover determines the theoretical advantage a powerful attacker might get. If we translate $0.888ns$ into distance, the maximum theoretical distance by which an attacker will be able to shorten its distance is about $12cm$.

We repeated this measurement 10 times, using the same setup. Figure 8 shows all 10 measured processing times along with their average value and a 95%




(a) Cable (b) Wireless

Figure 7: The delay of the prover's distance bounding radio extension. The top signal is measured at the reception antenna of the provers radio and is transmitted on channel $C_0$ at 3.5GHz. The bottom signal is measured at the transmission antenna and is being transmitted at the $C_2$ channel at 4.0GHz. The delay between them, and thus the prover's processing time is $0.888ns$.

confidence interval. We see from the figure that the processing time of the prover is stable between 0.8ns and 1ns.

Note that if the same setup would have been implemented in an integrated circuit, the signal path would be a lot shorter and consequently the processing time would have been smaller. We therefore do not claim that our prototype is the best that can be achieved, rather it shows the processing time that can be achieved using standard SMA components.

### 6.2 Wireless Implementation

Since distance bounding protocols are primarily useful in wireless environments, in this section we show that our prototype equally enables distance bounding using wireless communication (instead of wires). The basic construction of the prover is the same as in the wired setup, except that the prototype input and output are connected to antennas. The function generator that generates the verifiers signal and the oscilloscope used to measure the round trip time are likewise connected to antennas.

The result of the wireless implementation can be seen in Figure 7(b). Unfortunately we had to use SMA cables of about $1m$ to connect the antennas because of the way the antennas are mounted. In addition there was about $.1m$ between the transmission antenna and the receiving antenna. This results in a delay introduced by the cables and the space between the antennas referred to on Figure 7(b) as "antenna cable delay". The output of the prototype was passed through a high-pass filter and the input passed through a low-pass filter to prevent the transmitting antenna from feeding back into the receiving antenna. The oscilloscope used to measure the difference in arrival time also had filters to separate the ground truth signal, i.e., the signal coming directly from the function generator from the one being transmitted by the prototype. The filters allowed for a full duplex wireless channel to be created between our wireless prototype and the function generator and oscilloscope.

It should be noted that the channel switching mechanism of our prototype is ideal for a wireless implementation. Any wireless distance bounding protocol needs more than one channel (i.e., full duplex) in order to reply as fast as possible. Encoding the prover's reply in the choice of channel means that the solution is strait forward to apply without causing interference between the prover and verifier.

## 7 Related Work

Distance bounding, as a concept, was first proposed by Brands and Chaum in [3] who introduced techniques enabling a verifier to determine an upper-bound on the physical distance to a prover (as summarized in Section 2). In addition, they considered the case where the verifier also authenticates the prover in addition to establishing the distance bound.



Several optimizations and studies of distance bounding were subsequently proposed for wireless networks, including [28, 30, 5] and for sensor networks [18, 5, 27]. Distance bounding protocols have also been proposed in other contexts, e.g., for RFIDs [13, 10, 19] and ultra wide band (UWB) devices [17, 12].

In [23] the authors studied information leakage in distance bounding protocols. A mutual distance bounding protocol using interleaved challenges and responses was proposed in [31] and in [28] and [5] the authors investigated the use of distance bounding protocols for location verification and secure localization. Sastry, Shankar and Wagner [25] proposed the so-called "in-region verification" appropriate for certain applications, such as location-based access control. Collusion attacks on distance bounding location verification protocols where considered in [7, 6]. Ultrasonic distance bounding was used for access control [25] and for key establishment [32]. In [22] ultrasonic distance bounding was further used for proximity based access control to implementable medical devices. Other attacks have been proposed against distance bounding protocols in general. The so-called "late-commit" attacks where proposed in [14], where the attacker exploits the modulation scheme in order to manipulate the distance. Bit guessing attacks [8] that accomplish the same thing where also proposed. These attacks were further studied in practical implementations in [11].

Until now, most of the work done in this field has been theoretical. To our knowledge our work is the first to propose a realizable distance bounding protocol using radio communication, with a processing time at the prover that is low enough to provide a useful distance granularity.

## 8 Conclusion

We demonstrated that radio distance bounding protocols can be implemented to match the strict processing that these protocols require (i.e., that the prover receives, processes and transmits signals in $\leq 1ns$). This can be achieved using a specially implemented concatenation as the prover's processing function. Through this we showed that the use of processing functions which require that the prover demodulates (interprets) the verifier's challenge before responding to it, is not desirable or necessary for distance bounding. Finally, we showed that other processing functions such as XOR and the comparison function, that were used in a number of proposed distance bounding protocols, are not best suited for the implementation of radio distance bounding.

## References

[1] Thomas Beth and Yvo Desmedt. Identification tokens - or: Solving the chess grandmaster problem. In *CRYPTO '90: Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, pages 169–177, London, UK, 1991. Springer-Verlag.

[2] Colin Boyd and Anish Mathuria. *Protocols for authentication and key establishment*. Springer, 1998.

[3] Stefan Brands and David Chaum. Distance-bounding protocols. In *EUROCRYPT '93*, pages 344–359, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[4] Laurent Bussard and Walid Bagga. Distance-bounding proof of knowledge protocols to avoid terrorist fraud attacks. Technical report, Institut Eurecom, France, 05 2004.

[5] Srdjan Capkun and Jean-Pierre Hubaux. Secure positioning of wireless devices with application to sensor networks. In *IEEE INFOCOM*, 2005.

[6] Nishanth Chandran, Vipul Goyal, Ryan Moriarty, and Rafail Ostrovsky. Position based cryptography. In *CRYPTO '09: Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] Jerry T. Chiang, Jason J. Haas, and Yih-Chun Hu. Secure and precise location verification using distance bounding and simultaneous multilateration. In *ACM WiSec '09*, pages 181–192, New York, NY, USA, 2009. ACM.

[8] Jolyon Clulow, Gerhard P. Hancke, Markus G. Kuhn, and Tyler Moore. So near and yet so far: Distance-bounding attacks in wireless networks. In *Proceedings of the European Workshop on Security and Privacy in Ad-hoc and Sensor Networks (ESAS)*, 2006.

[9] Yvo Desmedt. Position statement in rfid s&p panel: From relative security to perceived secure. In *Financial Cryptography*, pages 53–56, 2007.

[10] Saar Drimer and Steven J. Murdoch. Keep your enemies close: Distance bounding against smartcard relay attacks. In *Proceedings of the USENIX Security Symposium 2007*, 2007.

[11] Manuel Flury, Marcin Poturalski, Panos Papadimitratos, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. Effectiveness of Distance-Decreasing Attacks Against Impulse Radio Ranging. In *3rd ACM Conference on Wireless Network Security (WiSec)*, 2010.

[12] S. Gezici, Zhi Tian, G.B. Giannakis, H. Kobayashi, A.F. Molisch, H.V. Poor, and Z. Sahinoglu. Localization via ultra-wideband radios: a look at positioning aspects for future sensor networks. *Signal Processing Magazine, IEEE*, 22(4):70–84, July 2005.

[13] Gerhard P. Hancke and Markus G. Kuhn. An rfid distance bounding protocol. In *SecureComm '05: Proceedings of the First International Conference on Security and Privacy for Emerging Areas in Communications Networks*, pages 67–73, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Gerhard P. Hancke and Markus G. Kuhn. Attacks on time-of-flight distance bounding channels. In *WiSec '08: Proceedings of the first ACM conference on Wireless net work security*, pages 194–202, New York, NY, USA, 2008. ACM.

[15] Y.-C. Hu, A. Perrig, and D. B. Johnson. Packet Leashes: A Defense against Wormhole Attacks in Wireless Networks. In *Proceedings of the IEEE Conference on Computer Communications (InfoCom)*, San Francisco, USA, April 2003.

[16] Yih-Chun Hu, Adrian Perrig, and David B. Johnson. Ariadne: a secure on-demand routing protocol for ad hoc networks. *Wirel. Netw.*, 11(1-2):21–38, 2005.

[17] J.-Y. Lee and R.A. Scholtz. Ranging in a Dense Multipath Environment Using an UWB Radio Link. *IEEE Journal on Selected Areas in Communications*, 20(9), December 2002.

[18] Catherine Meadows, Paul Syverson, and LiWu Chang. Towards more efficient distance bounding protocols for use in sensor networks. *Securecomm*, pages 1–5, Aug. 28 2006-Sept. 1 2006.

[19] Jorge Munilla, Andres Ortiz, and Alberto Peinado. Distance bounding protocols with void-challenges for RFID. Printed handout at the Workshop on RFID Security – RFIDSec 06, July 2006.

[20] National Space-Based Positioning, Navigation, and Timing Coordination Office. Global positioning system. `http://www.gps.gov/`.

[21] Maxim Integrated Products. An introduction to direct sequence spread spectrum communications. `http://www.maxim-ic.com/`, 2003.

[22] Kasper Bonne Rasmussen, Claude Castelluccia, Thomas S. Heydt-Benjamin, and Srdjan Čapkun. Proximity-based access control for implantable medical devices. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009.

[23] Kasper Bonne Rasmussen and Srdjan Čapkun. Location privacy of distance bounding protocols. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 149–160, New York, NY, USA, 2008. ACM.

[24] Qingchun Ren and Qilian Liang. Throughput and energy-efficiency-aware protocol for ultra-wideband communication in wireless sensor networks: A cross-layer approach. *IEEE Transactions on Mobile Computing*, 7:805–816, 2007.

[25] Naveen Sastry, Umesh Shankar, and David Wagner. Secure verification of location claims. In *WiSe '03: Proceedings of the 2nd ACM workshop on Wireless security*, New York, NY, USA, 2003. ACM.

[26] Patrick Schaller, Benedikt Schmidt, David Basin, and Srdjan Capkun. Modeling and verifying physical properties of security protocols for wireless networks. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 109–123, Washington, DC, USA, 2009. IEEE Computer Society.

[27] S. Sedighpour, S. Capkun, S. Ganeriwal, and M. Srivastava. Implementation of attacks on ultrasonic ranging systems, nov 2005.

[28] D. Singelee and B. Preneel. Location verification using secure distance bounding protocols. In *Mobile Adhoc and Sensor Systems Conference, 2005. IEEE International Conference on*, Nov. 2005.

[29] Nils Ole Tippenhauer and Srdjan Čapkun. Id-based secure distance bounding and localization. In *In Proceedings of ESORICS (European Symposium on Research in Computer Security)*, 2009.

[30] S. Čapkun, L. Buttyán, and J.-P. Hubaux. SECTOR: Secure Tracking of Node Encounters in Multi-hop Wireless Networks. In *Proceedings of the ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN)*, Washington, USA, October 2003.

[31] Srdjan Čapkun, Levente Buttyán, and Jean-Pierre Hubaux. Sector: secure tracking of node encounters in multi-hop wireless networks. In *ACM SASN '03*, pages 21–32, New York, NY, USA, 2003. ACM.

[32] Srdjan Čapkun and Mario Čagalj. Integrity regions: authentication through presence in wireless networks. In *WiSe '06: Proceedings of the 5th ACM workshop on Wireless security*, pages 1–10. ACM, 2006.

# The case for ubiquitous transport-level encryption

Andrea Bittau
*Stanford*

Michael Hamburg
*Stanford*

Mark Handley
*UCL*

David Mazières
*Stanford*

Dan Boneh
*Stanford*

## Abstract

Today, Internet traffic is encrypted only when deemed necessary. Yet modern CPUs could feasibly encrypt most traffic. Moreover, the cost of doing so will only drop over time. *Tcpcrypt* is a TCP extension designed to make end-to-end encryption of TCP traffic the *default*, not the exception. To facilitate adoption tcpcrypt provides backwards compatibility with legacy TCP stacks and middleboxes. Because it is implemented in the transport layer, it protects legacy applications. However, it also provides a hook for integration with application-layer authentication, largely obviating the need for applications to encrypt their own network traffic and minimizing the need for duplication of functionality. Finally, tcpcrypt minimizes the cost of key negotiation on servers; a server using tcpcrypt can accept connections at 36 times the rate achieved using SSL.

## 1 Introduction

Why is the vast majority of traffic on the Internet not encrypted end-to-end? The potential benefits to end-users are obvious—improved privacy, reduced risk of sensitive information leaking, and greatly reduced ability by oppressive regimes or rogue ISPs to monitor all traffic without being detected. In spite of this, end-to-end encryption is generally used only when deemed *necessary*, a small fraction of when it would be *feasible*.

Possible reasons for not encrypting traffic[1] include:

- Users don't care.
- Configuration is complicated and the payoff small (especially when connecting to unknown sites).
- Application writers have no motivation.
- Encryption (and key bootstrap) are too expensive to perform for all but critical traffic.
- The standard protocol solutions are a poor match for the problem.

We believe that each of these points either is not true, or can be directly addressed with well-established techniques. For instance, where users actually have control, they demonstrate that they do care about encryption. Four years ago only around half of WiFi basestations used any form of encryption [3]. Today it is rare to find an open basestation, other than ones which charge for Internet access.

It is clear, though, that application writers have little motivation: encryption rarely makes a difference to whether an application succeeds. Getting it right is difficult and time consuming, doesn't help time to market, and developers are hard-pressed to make the business case. For server operators, too, the process can be tedious. One reason people don't use SSL is that X.509 certificates are a mild pain both for the server administrator and, if the server administrator didn't buy a certificate from a well-known root CA, for users.

Even more important is the performance question. SSL is by far the most commonly deployed cryptographic solution, and it is expensive to deploy on servers. Where there is a need, such as for bank login or credit card payments, SSL is ubiquitous, but it is rarely used outside of web pages that are especially sensitive. The definition of "sensitive" has started to change, though; Google recently enabled SSL on all Gmail connections [25], ostensibly as a response to eavesdropping in China. In part this is possible today because cryptographic hardware has become comparatively inexpensive. This trend is set to continue; the most recent generation of Intel CPUs incorporate AES acceleration instructions [8], with the potential to significantly reduce the cost of software symmetric-key encryption.

Although symmetric-key encryption is unlikely to be

[1] Conspiracy theorists might suggest other reasons, but we won't discuss those here.

a problem, the conventional wisdom is still that it is too expensive to use public-key cryptography to bootstrap a session key for all network connections. Indeed our measurements show that a fully loaded eight-core (2 x Quad-core Xeon X5355) server can only establish 754 uncached SSL connections per second. In fact, this limitation is due to the way SSL uses public key algorithms rather than anything fundamental. We will show that much better server performance is possible with the right protocol design, in part by pushing costs to the client, which does not need to handle high connection rates.

Finally, there is the question of whether current encryption protocols are a sufficiently good match for applications that do not currently use encryption. We believe they are not, for reasons we shall highlight throughout the paper. However, we will describe a subtly different protocol architecture that we believe is a much better fit to the majority of applications. This is not rocket science; it may even be considered obvious. But we believe it makes a huge difference to the deployability of encryption and consequently of authentication in the real world.

### 1.1 Getting the Architecture Right

All the commonly deployed network encryption mechanisms incorporate authentication into the protocol, even if, like WPA, it is as simple as requiring out-of-band password exchange. Indeed this is the obvious way to engineer things; without authentication, it is not possible to determine if your encrypted channel is with the desired party or with a man-in-the-middle. However, we believe that this is fundamentally the wrong design choice.

Encryption of a network connection is a general purpose primitive; regardless of the application, the goal is to prevent eavesdroppers from learning the contents of communications. MACing of packets in a network connection is also a general purpose primitive; no application wants to accept forged or maliciously modified packets. Authentication, however, is not general purpose. The mechanism used for authentication and the information needed to perform that authentication are application-specific. In practice, protocols blur this distinction between general purpose encryption/integrity and special purpose authentication. This has two consequences:

- It tends to encourage inappropriate authentication mechanisms. For example, using SSL to connect to a bank, then simply handing the user's password to the bank, when it is known that people commonly re-use passwords across sites.

- It makes it hard to integrate mechanisms low enough in the protocol stack to really be ubiquitous. For example, adding SSL to an application requires modifying the source code and, potentially, extending its application-layer protocol in a backwards compatible way.

To enable encryption and integrity checking in a general way for all legacy TCP applications[2], this functionality must be *below* the application layer. However it cannot be done *cleanly* any lower than the transport layer because this is the lowest place in the stack that has any concept of a conversation. There is also the practical consideration that encrypting below the transport layer will prevent NAT traversal. The clear implication is that embedding encryption and integrity protection *into* TCP would provide the right general-purpose mechanism; in fact, because TCP includes a session establishment handshake, this is simple to do in a backward-compatible way.

To establish session keys in a general way, TCP-level encryption should be divorced from higher level authentication mechanisms. This suggests the use of ephemeral public keys to establish session keys. Such a mechanism, enabled by default, would provide protection against passive eavesdroppers for all TCP sessions, even for legacy applications. We are not the first to suggest such "opportunistic" encryption. Our goal, though, is to provide not just encryption and integrity protection, but also a firm foundation upon which higher-level authentication mechanisms can build. With the right architecture, a diverse set of authentication mechanisms can be devised, each suitable to its own application.

The end point we hope to establish is that all TCP sessions (and SCTP and DCCP, though we don't discuss these further here) are protected against passive eavesdroppers, and that all applications that require authentication should, as a side effect, enjoy protection against active man-in-the-middle attacks, all without duplication of effort. Ideally, an eavesdropper cannot tell from watching the traffic which encrypted sessions will be authenticated.

In this paper, we describe *tcpcrypt*, our implementation of TCP-level encryption. Although the idea is simple, the details really matter, as we will show. We have validated our design by building two implementations, one a Linux kernel module, the other a user-space process using divert sockets. The latter allows use of tcpcrypt on Linux, FreeBSD, and MacOS X without modifying the kernel. Both implementations show excellent performance; we will demonstrate that this is no longer the factor preventing ubiquitous network encryption. We have also implemented application-level authentication protocols that use tcpcrypt to bootstrap authentication. These include X.509 certificate-based authentication, fast password-based *mutual* authentication, and PAKE. Our X.509-based authentication provides security equivalent to SSL, but uses batch-signing to run 25 times faster. Moreover, we have implemented X.509 authentication inside the OpenSSL library in a way that preserves the same API and cleanly falls back to vanilla SSL when appropriate. Thus, to take advantage of tcpcrypt in SSL-enabled applications requires only a library update.

[2]The vast majority of Internet applications use TCP.



## 2 Cryptographic design

The goal of tcpcrypt is to enable the best communications security possible under a wide range of circumstances. In the absence of any authentication, when users browse unknown servers, they should enjoy protection from passive eavesdropping. Though active network attackers may still intercept and monitor communications (there are also legitimate reasons for this, such as transparent proxies and intrusion detection systems), it should be possible to detect such behavior both during communications and afterward. Thus, tcpcrypt should virtually eliminate the possibility of widespread eavesdropping unbeknownst to a user population.

When an application performs any kind of endpoint authentication, it must be able to leverage tcpcrypt to obtain stronger protection of session data. For instance, given a server-side X.509 certificate, the client should be assured of the confidentiality of the data it transmits and the integrity of the data it receives. Any time a user types a password, it should be possible to ensure the confidentiality and integrity of all data sent in either direction.

In all cases, when tcpcrypt achieves confidentiality, it should also provide forward secrecy. As a final goal, tcpcrypt should affect performance as little as possible. Thus, the protocol is designed to minimize the number of cryptographic operations and extra round trips, subject to the limitations of needing to interoperate with legacy end hosts and middleboxes.

### 2.1 Key exchange protocol

Key exchange is the biggest challenge to tcpcrypt's performance. Forward secrecy requires a pair of hosts to exchange a secret using an ephemeral public key or Diffie-Hellman key exchange the first time they communicate. These operations are far more costly than establishing a TCP connection, but the cost can be asymmetric. For example, a single core of the server in Section 6 can perform 12,243 encryptions/sec with a 2,048-bit RSA-3 key, but only 97 decryptions/sec.

Servers typically communicate with more peers than clients do, so it makes sense for clients to shoulder most of the cost of key exchange. Thus, by default, tcpcrypt performs the expensive decryption at the client (though for generality, servers may opt to reverse the protocol).

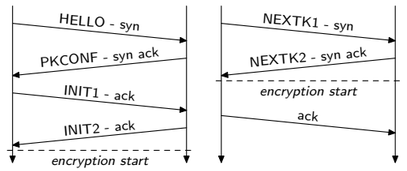



Figure 1: Tcpcrypt connection establishment with key exchange (left) and session caching (right).

Subsequent connections between the same two hosts can use *session caching* to avoid any public key operations at all, thereby ensuring that, for instance, an active-mode FTP server need not perform RSA decryptions.

The initial key exchange works as follows. Each machine $C$ has an ephemeral public key, $K_C$. When $C$ connects to a server $S$ for the first time, $C$ chooses a random nonce, $N_C$; $S$ chooses a random secret, $N_S$; the two exchange the following messages, also shown in Figure 1:

$C \to S$: HELLO
$S \to C$: PKCONF, pub-cipher-list, [cookie]
$C \to S$: INIT1, sym-cipher-list, $N_C$, $K_C$, [cookie]
$S \to C$: INIT2, sym-cipher, ENCRYPT $(K_C, N_S)$

Here pub- and sym-cipher-list are used to negotiate cryptographic algorithms. The optional cookie is a SYN-cookie that must be echoed by the client to make it harder for packets from forged source addresses to trigger any public-key cryptographic operations in the server. This trade-off is at the discretion of the server; if TCP's 32-bit initial sequence number (ISN) provides enough protection against forged packets, the option space may be deemed better used for other purposes.

$K_C$ specifies the public key cipher and a pseudo-random function, used below. Quantities from this protocol are then combined into a series of "session secrets" with a Collision-resistant Pseudo-random Function, CPF (currently HMAC):

$$\text{ss}[0] \leftarrow \text{CPF}\big(N_S, \{K_C, N_C, \text{cipher-lists, sym-cipher}\}\big)$$
$$\text{ss}[i] \leftarrow \text{CPF}\,(\text{ss}[i-1], \texttt{TAG\_NEXT\_KEY})$$

If $\text{ISN}_{C,i}$ and $\text{ISN}_{S,i}$ are TCP's initial sequence numbers on the client and server for session $i$, the two sides then compute a *master secret* as follows:

$$\text{mk}[i] \leftarrow \text{CPF}\,(\text{ss}[i], \{\texttt{TAG\_KEY}, \text{ISN}_{C,i}, \text{ISN}_{S,i}\})\,.$$

Finally, the two sides use $\text{CPF}(\text{mk}[i], x)$ on various constants $x$ to generate encryption and MAC keys (a common technique). From this point on, all further segments in the TCP connection are cryptographically protected.

Note that this full key exchange is only needed for the first connection between two hosts. Hosts can cache ss$[i]$

for the largest $i$ used till that point. Subsequent connections between the same two hosts can use this to derive new symmetric keys, thereby avoiding any further public key cryptography and the latency of the full handshake.

## 2.2 Authentication Hooks

To gain stronger benefits from tcpcrypt, applications must be able to make statements about a connection—*e.g.,* "All data you read from this connection is sent by user U's browser," or "Any data you write to this connection can be decrypted only by server Y." To make such statements, one must specify what is meant by "this connection" in a way that cannot be interpreted out of context. Tcpcrypt accomplishes this through *session IDs*. A new *getsockopt* call returns a session ID, $\mathrm{sid}[i]$, computed from the connection's session secret $\mathrm{ss}[i]$ as follows:

$$\mathrm{sid}[i] \leftarrow \textsc{cpf}\,(\mathrm{ss}[i], \texttt{TAG\_SESSION\_ID})$$

If both ends of a tcpcrypt connection see the same session ID, then with overwhelming probability an attacker cannot eavesdrop on or undetectably tamper with traffic—*i.e.,* there has not been a man-in-the-middle attack. Two properties facilitate verification of session IDs. First, they need not be kept secret. Second, with overwhelming probability they are unique over all time, even if one end of a connection is malicious. Hence, a cryptographically endorsed session ID can only ever authenticate a single tcpcrypt connection. In Section 4 we discuss different ways applications can leverage session IDs.

## 2.3 Proof of Security

To increase confidence in tcpcrypt, we provide a semi-formal proof of its security. We assume that the adversary has complete control over the network, and nearly complete control over the users. It can choose when and to whom users attempt to connect, and what data they send, and can delay, drop, modify, and forge packets arbitrarily. Furthermore, since the session IDs $\mathrm{sid}[i]$ are not secret, we assume that the adversary knows them. We do not model malicious machines here, as the adversary can emulate as many of these as it wants. We do not model compromised machines because of space constraints. When we write "client" or "server" in this discussion, we mean a legitimate client or server.

We guarantee the security of tcpcrypt connections only when the session IDs match. In this case, the guarantee is fairly strong:

**Definition 2.1** (Security guarantees)**.** *Suppose that users $U_1$ and $U_2$ complete the tcpcrypt protocol on sockets $S_1$ and $S_2$, and arrive at sessions with the same session ID. Then the following guarantees hold:*

- *The adversary has not tampered with $U_1$ and $U_2$'s cipher suite choices. Assuming they have chosen a secure cipher suite:*
- *Any packet sent by $U_1$ on socket $S_1$ (or by $U_2$ on $S_2$) gives no information to the adversary other than its length and timing.*
- *If, after TCP reassembly, $U_2$ receives a sequence of segments $p_1, \ldots, p_n$, then $U_1$ sent those segments in that order (and no segments before them), and similarly for segments received by $U_1$.*

We will show that, unless the adversary has broken the underlying cryptographic primitives, its probability of violating this guarantee is very small. Specifically:

**Theorem 2.1** (Security of tcpcrypt)**.** *Suppose that an adversary $\mathcal{A}$ can violate the tcpcrypt security guarantee with probability $\epsilon$. Suppose that it uses $m$ machines in its attack, and begins at most $c$ connections in total. Then there are five simple modifications of $\mathcal{A}$, running in about the same time as $\mathcal{A}$, which aim to do the following things:*

- *Find a collision in* CPF.
- *Break the pseudorandomness of* CPF.
- *Break the public-key cipher.*
- *Break the MAC.*
- *Break the symmetric cipher.*

*The sum of their probabilities of success is at least*

$$\epsilon - 3c^2/2^{k+1}$$

*where $k \approx 256$ is the minimum of the min-entropy of a public key, or the length in bits of $N_S$ or $N_C$.*

*Proof.* Define $\textsc{next}(k) := \textsc{cpf}(k, \texttt{TAG\_NEXT\_KEY})$. Suppose that $U_1$ and $U_2$ have the same sid, and that for $U_1$ it is $\mathrm{sid}[i]$ for some $i$, where:

$$\mathrm{ss}[0] = \textsc{cpf}\,(N_S, \{K_C, N_C, \text{cipher-lists, sym-cipher}\})$$
$$\mathrm{sid}[i] = \textsc{cpf}\left(\textsc{next}^i(\mathrm{ss}[0]), \texttt{TAG\_SESSION\_ID}\right)$$

Because everything passed to CPF has a unique parse, the sid must have been computed by $U_2$ in the same way—and in particular with the same values of $N_S, N_C, K_C$, the same cipher suite lists and the same cipher choice—or else the computation contains a hash collision. What is more, the $N_S, N_C$, and $K_C$ values are chosen at random, and so with probability at least $1 - 3c^2/2^{k+1}$ they are unique. For the rest of the proof, assume that this is the case.

Now, each of $U_1$ and $U_2$ is either a client or a server. Because their $K_C$, $N_C$ and $N_S$ values match, they can't both be clients or both be servers; without loss of generality, say $U_1$ is the client (which generated $K_C$ and $N_C$), and $U_2$ is the server (which generated $N_S$).

We will next show that this $N_S$ remains secret. We first replace $\textsc{encrypt}(K_C, N_S)$ with an encryption of zero (but the client still decrypts it to $N_S$). If the adversary notices this, then it has broken the public-key cipher. After this change, $N_S$ is only used as a key to CPF. Furthermore, CPF is evaluated on $N_S$ only once by $U_2$ and once by $U_1$, with a nonce $N_C$ in the other argument; if the adversary replays $\textsc{encrypt}(K_C, N_S)$, then $\textsc{cpf}(N_S, \cdot)$ will be called with different nonces. Because CPF is pseudorandom, we can replace its outputs $\mathrm{ss}[0]$ with independent random values; if the adversary notices this, then it has broken CPF. Continuing in this manner, we can replace $\mathrm{ss}[i]$, $\mathrm{mk}[i]$, $\mathrm{sid}[i]$ and the encryption and MAC keys with random values, and the adversary will not notice this, either.

If the initial sequence numbers do not match, the client and server will arrive at different (secret, random) MAC keys, and so as long as the MAC is unforgeable, neither will accept any packets at all. Otherwise since every packet is MACed with associated data that includes the 64-bit extended sequence number, they must be received unmodified and in order. Finally, if the symmetric cipher is secure against chosen-plaintext attacks, the only information that the adversary can learn about a segment is its length and timing. This completes the proof. □

# 3 Integration with TCP

Integrating tcpcrypt into TCP posed a number of challenges ranging from the basic to the baroque. First, we have to extend TCP in a backwards compatible way. If a tcpcrypt client connects to a tcpcrypt server, encryption should be enabled by default, but if it is a legacy server, the session must fall back to regular TCP behavior.

The same issue applies with middleboxes. Tcpcrypt must work through NATs, so it cannot protect the TCP ports. Tcpcrypt must also work correctly when faced with firewalls that do not understand the tcpcrypt extensions. For an example of how broken firewalls have inhibited innovation, we need look no further than Explicit Congestion Notification (ECN). ECN should be harmless to deploy—it uses TCP options in the handshake to negotiate the capability, then uses two bits from the old IP Type-of-Service field to indicate congestion, and finally signals this in feedback using a previously reserved TCP flag. ECN is built into all the main modern operating systems, but is disabled by default. This is because a small number of home gateway/firewall boxes crash when they see the reserved TCP flag set to one.

This has taught us to avoid protocol changes to TCP that are not carried in TCP options. Firewalls might drop unknown options, or might completely drop packets with unknown extensions; a TCP extension needs to be robust to either and correctly fall back to regular TCP behavior.

Finally we risk being hoisted by our own petard. Traffic normalizers [9], as implemented in `pf` [10] and some other firewalls, enforce conservative rules on protocol behavior and consistency. This limits design flexibility.[3]

## 3.1 Initial TCP Handshake

Ideally the key exchange for tcpcrypt would be performed in TCP's three-way connection setup handshake, as this would add no additional network latency to establishing encrypted sessions. We can't quite achieve this for the first connection between two hosts—rather, we require adding information to the first four packets of the session, as shown in Figure 1. To be backwards compatible with regular TCP, any data we can add to the SYN and SYN/ACK packets must fit within the TCP options field, which is limited to 40 bytes, some of which are required to negotiate other TCP functionality. This requires HELLO and PKCONF to be small. HELLO requests encryption; PKCONF acknowledges the use of encryption and states the list of public key ciphers that can be used for the subsequent key exchange. Receipt of a SYN/ACK without PKCONF causes fallback to vanilla TCP.

The INIT1 message cannot be small, as it must contain the client's public key. The public key cannot fit into an option, so instead we re-purpose the data portion of one packet in each direction to carry it. The data payload is only co-opted in this way after tcpcrypt negotiation has succeeded, which ensures that key data never accidentally gets passed to applications by legacy TCP stacks. INIT2 is sent in response to INIT1 in the same way.

We use a single TCP "CRYPT" option; HELLO, PKCONF, INIT1, and INIT2 are suboptions of CRYPT. This reduces the use of scarce TCP option numbers, but more importantly it ensures that if a middlebox is going to remove one option, it should remove them all. If either host receives a TCP segment without a CRYPT option during session establishment, tcpcrypt falls back to vanilla TCP. This ensures interoperability with non-tcpcrypt-aware stacks and middleboxes that strip out unknown options. Applications can test whether tcpcrypt is used by calling *getsockopt* to request the *session ID*, which returns an error on downgraded connections.

Tcpcrypt also incorporates a re-keying mechanism, allowing session keys to evolve later in the connection to avoid using a single set of session keys for too long.

## 3.2 Session Caching

Applications such as the Web often establish more than one TCP connection between the same pair of hosts in rapid succession. When they do this, the amount of data

[3] One of us sometimes regrets writing the Normalizer paper.

transferred per connection can be quite small—often a few KBytes. If we have to pay the full cost of running the public key operations to establish these short-lived sessions, tcpcrypt can become a bottleneck. Fortunately we can use the same solution as SSL—cache the cryptographic state from one TCP connection and use it to bootstrap subsequent connections.

To do this we use two more CRYPT suboptions, NEXTK1 and NEXTK2, also shown in Figure 1. We cannot depend on the IP address in the SYN packet to locate the correct state because the client may have moved, or a different client may have acquired the DHCP lease used by a previous client. Thus NEXTK1 contains nine bytes of the next session ID, sid$[i+1]$. This allows the server to verify that it has the correct cached state before using it to enable encryption. It also makes it hard for DoS attackers to flush the server's cache by spoofing packets. In the event of a cache miss, the server returns PKCONF and the protocol falls back to ordinary key exchange.

### 3.3 Protocol and Data Integrity

Unlike SSL, one of tcpcrypt's goals is to provide integrity protection for the TCP session itself, defending against attacks that might reset the connection [5], insert data into it, or otherwise interfere with its progress [14]. To do this, tcpcrypt adds a MAC option to every TCP packet after the INIT1/INIT2 exchange. Packets received with an incorrect or missing MAC are silently dropped.

This MAC option authenticates a segment's payload as well as a pseudo-header comprising most of the TCP header fields and options, as shown in Figure 2. We need to be pragmatic about which fields are covered by the pseudo-header. The TCP ports cannot be covered, as NATs re-write them. The MAC option is zeroed out in the pseudo-header, since it cannot authenticate itself.

Replay attacks could present a potential issue when TCP's sequence space wraps. Instead of sequence and acknowledgment numbers, the pseudo-header contains implicitly extended 64-bit values that cannot wrap. The acknowledgment number is fed separately into the MAC value, with a technique from [15], so as to improve the efficiency of retransmissions (which often acknowledge a different packet from the original).

Extended sequence numbers also solve the problem that PAWS [13] was intended to solve, so an encrypted TCP session might omit the timestamp option. This frees up eight bytes of option space; if we use a 64-bit MAC then tcpcrypt will use no more option space than most modern TCP implementations. This is particularly relevant for high performance, because when TCP's window is large it benefits from the robustness provided by Selective Acknowledgments (SACK) [19], and we do not wish to reduce their effectiveness.

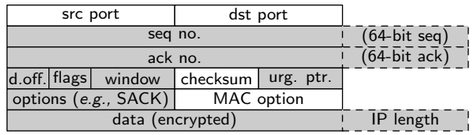



Figure 2: A data packet using tcpcrypt. Dashed quantities are not transmitted by TCP though included in the MAC, along with shaded fields.

More subtly, we need to be careful about middleboxes that modify packets. If an implementation does send the timestamp option, tcpcrypt will normalize it to zero in the pseudoheader, as OpenBSD's pf [10] modulates its value. All the other options that are commonly modified occur only in the SYN or SYN-ACK, so do not present a problem. Tcpcrypt does provide a secure timestamp-like suboption to CRYPT called SYNC. SYNC is covered by the MAC, but fuzzes the clock to avoid the reasons for which pf needs to modulate the timestamp's value. Moreover, the SYNC option is only required for keepalive packets and during re-keying when the connection is otherwise idle. In both cases there is no need for SACK blocks, so the option space is less precious.

Packets with the TCP RST bit set present the final challenge. For full protection, after session establishment we would prefer to drop RST packets that do not contain a valid MAC option. However, RST is TCP's mechanism for informing one side of a connection that the other side no longer has any state for the connection. Under such circumstances it is impossible for a legitimate host to generate a RST packet with the MAC option. Tcpcrypt's default behavior is to reset the connection when receiving a RST with no MAC, so long as it passes the OS's sequence number validity checks. However, some applications (notably BGP routing) have a much stronger requirement to protect against connection resets. For these applications we support a setsockopt that mandates RST packets carry a valid MAC. Such connections will take a long time to time out if one side loses state; however, applications such as BGP and SSH that might require such protection also typically use application-level keepalives to detect liveness and so tear down stale connections.

### 3.4 Application Awareness

Tcpcrypt serves a dual role: for legacy applications it protects against passive eavesdroppers; for tcpcrypt-aware applications it enables stronger protection, as we will discuss below. However, it is important to avoid a duplication of functionality.

Consider a tcpcrypt-aware web browser on a tcpcrypt-capable host that wishes to make an authenticated connection to a web server. The browser might prefer tcpcrypt because of the availability of better password authentication methods, but only if the web server also supports it. Otherwise, it wishes to fall back to SSL.

A potential problem occurs when the client connects to a legacy web server process running on a tcpcrypt-capable host. Under such circumstances we do not wish to use both unauthenticated tcpcrypt and authenticated SSL encryption, which would be the default behavior. Rather, the web browser wishes the tcpcrypt negotiation in the SYN exchange to fail unless *both* the host and the web server process can use the tcpcrypt-based authentication.

To get this correct fallback behavior, the HELLO option includes a "*Mandatory Application-Aware*" bit. When set, this bit indicates to the server that it *must not* enable tcpcrypt encryption unless the server application has informed the stack that it is tcpcrypt-aware. The process uses a `setsockopt` on the listening socket to do this. Our enhanced SSL implementation that uses this mechanism is described in Section 5.3.

Tcpcrypt also includes a second "*Advisory Application-Aware*" bit in both the HELLO and PKCONF options. This is used for each side to indicate to the other that the application is tcpcrypt-aware. This is used when applications want to perform authentication over tcpcrypt if the other side is also tcpcrypt-aware, but where it is not necessary to fall back to an unencrypted session if the other side is not tcpcrypt-aware. For example, many websites with low security requirements use HTTP Digest authentication. Such websites can still use HTTP Digest authentication over tcpcrypt (though we would not advise it), but if both the client and server applications are tcpcrypt-aware, it would be possible to drop in CMAC-based mutual authentication instead. However, the client needs to know that the server can do this before sending the HTTP request, and the "*Advisory Application-Aware*" bit provides this information. It is set via a `setsockopt` before calling `connect` and retrieved at the other side via `getsockopt` after the connection handshake completes.

## 4 Authentication examples

User authentication is an area in which there exist simple and well-known techniques qualitatively superior to those in widespread use. For instance, websites typically request passwords be sent straight to the server. As a result, we see many successful phishing attacks. Almost all of these attacks could very easily be defeated with known techniques, were it not for issues of backwards compatibility in protocols and user interfaces. Thus, there are strong incentives to make improvements to authentication in the web and other applications.

To realize this shift to better authentication protocols we need innovation in user-interface design. Currently, HTTP digest authentication, while better than plaintext passwords, is seldom used because web developers shun browsers' ugly gray popup boxes. The challenge is to allow some aesthetic control by web sites while simultaneously ensuring password entry is unambiguously differentiated from web forms (or anything else accessible by JavaScript). Tcpcrypt itself obviously cannot improve user interfaces; the aim is to ensure that when improvements do happen, they can easily be integrated with tcpcrypt to provide security against active attackers.

The hook tcpcrypt provides to application-level authentication is the session ID. This section gives a few examples of how session IDs can be used, assuming the ability to display certificate names and to input passwords from a user securely. Though these examples require modifications to applications, such enhancements can be deployed incrementally using tcpcrypt's Application-Aware bits described in the previous section.

Note that the prevalence of weak authentication makes for some very low-hanging fruit. We do not claim these obvious and well-known fixes as contributions. Nor do we mean to imply that these techniques would not work with application-layer traffic encryption were we to enhance SSL. Our point is merely to illustrate the generality of the session ID abstraction and to help substantiate our claim that tcpcrypt provides encryption as a general building block suitable for a wide range of applications.

The key properties we rely on are that 1) if both ends of a connection see the same session ID, then the session data's confidentiality and integrity are ensured, and 2) session IDs are unique over all time with overwhelming probability, even when one end of a connection is malicious.

### 4.1 Certificate-based authentication

One common basis for server authentication is certificates, such as the X.509 certificates employed by SSL. (This model may become even more prevalent if DNSSEC gains widespread deployment.) In this model, each server $S$ has a long-lived public key, $K_S$, certified by a trusted authority to belong to a particular common name and organization. The common name or organization can then be presented to the user to inform her of whom she is communicating with.

Certificates permit a trivial authentication protocol:

$$S \to C\colon\ K_S, \text{Certificate}, \textsc{Sign}\left(K_S^{-1}, \text{Session ID}\right)$$

The server simply signs the session ID, thereby proving it owns one end of the connection, ensuring confidentiality

of messages sent by the client and integrity of those sent by the server.

The problem with the above protocol is the cost of the SIGN function, which can be comparable to public-key decryption. The cost for the server to compute such a signature for every new client would be comparable to setting up an SSL connection, which is one of the factors dissuading people from using SSL ubiquitously today. While there do exist some faster signature schemes (*e.g.,* [7]), the certificate authorities may not be willing to endorse non-standard algorithms.

Fortunately, there is a better approach. Heavily loaded servers can amortize the cost of a single signature over many sessions by signing a batch of session IDs. Session IDs are not secret, so disclosing a batch of them to each client is not a problem.

Once a single session has been authenticated, the same pair of machines can use the existing connection to bootstrap authentication of other sessions using only symmetric cryptography. For instance, they can exchange a MAC key and use it to authenticate future session IDs.

## 4.2 Weak password authentication

Often two connection endpoints share a secret. For instance, a user may remember a password, and a server may store some secret derived from the password. Today, all too often passwords simply authenticate the user to the server and not vice versa. As a basic principle, if we deploy new authentication mechanisms, *any time a user types a password, it should mutually authenticate the client and server to each other*. There is simply no reason ever to use a password to authenticate only one endpoint of a communication. Even if the other end is a server with an X.509 certificate, the certificate may have been fraudulently obtained, or it may be for a "typo" domain name similar enough to the desired one that the user doesn't notice the error.

When a server, $S$, is under severe performance constraints, it can perform password authentication using symmetric cryptography. For instance, $S$ may store the secret hash value of a user's password, $h = H(\text{salt}, \text{realm}, \text{password})$; a client $C$ can query $S$ for the non-secret salt, then compute $h$ from a user-supplied password. Section 6 benchmarks the following trivial authentication protocol for such settings:

$$C \to S\text{: } \text{MAC}\left(h, \texttt{TAG\_CLIENT} \| \text{Session ID}\right)$$
$$S \to C\text{: } \text{MAC}\left(h, \texttt{TAG\_SERVER} \| \text{Session ID}\right)$$

This protocol is no more costly or hard to implement than digest authentication [6] (in fact, possibly easier, as it requires no randomness beyond that already reflected in the Session ID). Yet it provides better guarantees, namely *mutual* authentication of $S$ to $C$ as well as integrity and confidentiality of all session data. The protocol assures both $C$ and $S$ that the other end of the connection knows $h$. Such a guarantee is different from and complements that provided by certificates—*i.e.,* that a server owns a particular domain name. Domain-name certificates offer important protection in many contexts, but this session-ID-based protocol offers protection even when users do not remember the correct domain name.

We note that even if an attacker hijacks DNS to impersonate $S$, our protocol is resistant to phishing for users with good passwords. The protocol can be viewed as endorsing the session ID with $h$; since session IDs are unique over time, the attacker may obtain $\text{MAC}(h, \texttt{TAG\_CLIENT} \| \text{Session ID})$, but this value is meaningless in the context of any other connection.

Unfortunately, while the above protocol would be categorically superior to plaintext passwords and digest authentication, we still do not advocate using it except for servers on which stronger authentication would require too much CPU time. The problem is that an attacker who impersonates the server to obtain the first message can then mount an offline dictionary attack on the password, leveraging the single message exchange to guess arbitrarily many passwords. Such an attack may be detectable if the attacker cannot crack the password in time to mount a transparent man-in-the-middle attack—but people are used to clicking reload sometimes when web sites fail and will not be concerned by a single connection failure.

## 4.3 Strong password authentication

Fortunately, as detailed in Section 6, any site that can afford to use SSL today can afford to use a strong password authentication scheme with tcpcrypt. Here we give a simple example of a Password-Authenticated Key-Exchange (PAKE) protocol that that, while considerably more expensive than the previous weak protocol, can nonetheless be implemented with far less overhead than SSL imposes today.

We use a protocol termed $\texttt{PAKE}_2^+$ in [4]. The protocol relies on several system-wide parameter choices: a group $\mathbf{G}$ of prime order $q$ (on which the computational Diffie Hellman problem is hard); a generator $g$ of $\mathbf{G}$; two randomly-chosen elements of $\mathbf{G}$, $U$ and $V$; two cryptographic hash functions, $H_0$ and $H_1$, mapping strings to elements of $\mathbf{Z}_q$; and finally, another hash function, $H$, onto bit strings the size of a MAC key. At the time a user registers for an account, her client computes:

$$\pi_0 = H_0(\text{password, user name, server name})$$
$$\pi_1 = H_1(\text{password, user name, server name})$$
$$L = g^{\pi_1}$$

The server stores $\pi_0$ and $L$, but never sees $\pi_1$. To authenticate a session, the client chooses a random element $\alpha \in \mathbf{Z}_q$ and the server chooses a random element $\beta \in \mathbf{Z}_q$. The two then engage in the following protocol:

$$C \to S\text{: } g^{\alpha} U^{\pi_0}$$
$$S \to C\text{: } g^{\beta} V^{\pi_0}$$

At this point, both sides compute $g^{\alpha\beta}$. They can do this by computing either $U^{-\pi_0}$ or $V^{-\pi_0}$ and using it to revert to a regular Diffie-Hellman key exchange. Then both sides compute $g^{\pi_1\beta}$. The client can do this because it knows $g^{\beta}$ and $\pi_1$. The server can do this because it knows: $L = g^{\pi_1}$ and $\beta$. Finally, both sides compute:

$$h = H\left(\pi_0, g^{\alpha}, b^{\beta}, g^{\alpha\beta}, g^{\pi_1\beta}\right)$$

Using $h$ they complete the password authentication protocol of the previous section, but now the order of messages doesn't matter (the client and server can each transmit one of these messages before receiving the other to reduce latency):

$$S \to C\text{: } \text{MAC}\left(h, \texttt{TAG\_SERVER} \| \text{Session ID}\right)$$
$$C \to S\text{: } \text{MAC}\left(h, \texttt{TAG\_CLIENT} \| \text{Session ID}\right)$$

While this protocol is considerably more expensive than the one in the previous section, it has the benefit of protecting users with weak passwords; each guess at the password requires a separate network interaction with a party that knows either the password or $\pi_0$ and $L$. Moreover, the protocol is still cheaper than SSL (even combined with tcpcrypt key negotiation). Therefore, we believe it is suitable for use in any application that uses both passwords and SSL.

It is an open question whether we can design password authentication protocols that are highly efficient at the server and offload most of the work to the client. However, should we devise such protocols, they can be deployed after the fact, without modification to tcpcrypt itself. The session ID abstraction nicely separates tcpcrypt's confidentiality and integrity properties, which are solved problems, from authentication, where further innovation may be needed.

# 5 Implementation

To validate the protocol design and verify its performance, we implemented tcpcrypt in the Linux kernel. We also implemented tcpcrypt as a user-space daemon using divert sockets; this allows tcpcrypt to be deployed easily without requiring any kernel changes. Finally we implemented a range of application authentication mechanisms over tcpcrypt.

## 5.1 Linux kernel implementation

Our kernel implementation of tcpcrypt consists of a 4,000-line loadable module and 70 lines added to the core Linux 2.6.32 kernel to add the necessary hooks. For RSA support, we ported OpenSSL v0.9.8l to the Linux kernel. This required about 400 lines of glue code to export RSA as a Linux crypto module. We also exposed OpenSSL's SHA1 as we found it to perform twice as fast as Linux's implementation.

During the implementation, it became clear that tcpcrypt is incompatible with TCP segmentation offloading, as supported in some modern NICs. As tcpcrypt has to copy the packet to memory to encrypt the data and compute the MAC, segmenting it during this process does not add significant overhead. However, a server running so close to its performance limits that it requires segmentation offloading would likely want to disable tcpcrypt.

## 5.2 Portable userspace implementation

Our userspace tcpcrypt implementation uses divert sockets to access TCP packets entering and leaving the host. Firewall rules select the packets to be diverted, leaving the kernel unchanged. FreeBSD's NAT (`natd`) is implemented this way. The main advantages of this approach are portability and ease of deployment. Our code is 7,000 lines. We have tested it on MacOS X, FreeBSD and Linux.

The userspace implementation is obviously slower than the native kernel implementation, but it is ideal for early deployment without support from OS vendors. If tcpcrypt is successful and ships in major operating systems, it will still be a long time before older hosts are upgraded. The userspace implementation provides a good interim solution. It can also be run on middleboxes such as firewalls or home gateways to protect traffic to and from legacy local hosts against passive eavesdropping.

The userspace implementation is more complicated than the kernel one as it must track connections, duplicate much of TCP's state machine, calculate checksums again, and rewrite sequence and acknowledgment numbers since we use some bytes of the payload for INIT messages. In SYNs the MSS is reduced to allow space to add the MAC to subsequent packets. In addition, the sending of application data must be delayed until the tcpcrypt handshake completes, which we do by modulating the receive window. Finally, we implement IPC calls to provide the equivalent of *getsockopt*, so the application can extract the session ID to perform authentication.

## 5.3 Integrating tcpcrypt and OpenSSL

If tcpcrypt were enabled by default, then an SSL connection between two tcpcrypt hosts would duplicate effort doing both tcpcrypt and SSL key exchange and encryption. Tcpcrypt's *Mandatory Application-Aware* bit avoids this duplication. To verify this mechanism and to compare the full performance of Apache running SSL-over-tcpcrypt using batch-signing to that of vanilla SSL, we implemented tcpcrypt support within the OpenSSL

v0.9.81e library. We did not modify OpenSSL's API or require applications to set specific parameters to gain the benefits of tcpcrypt and batch-signing—our library is a drop-in replacement for OpenSSL.

Our implementation uses the tcpcrypt `setsockopt` to notify the kernel that the application supports tcpcrypt, setting the *Mandatory Application-Aware* bit during the handshake. After the TCP handshake, either the session is encrypted and both sides support tcpcrypt-based authentication, or the connection has fallen back to vanilla TCP. The library code then queries with `getsockopt` to get the session ID. If this returns an error, it falls back to SSL's handshake, otherwise it batch-signs the session ID and sends it to the client.

We modified OpenSSL's BIO layer to call the necessary `setsockopt` for setting the application bit. The SSL layer, *i.e.,* `SSL_accept` and `SSL_connect`, then deals with the signatures. Thus, so long as the application uses the BIO API, no change to the application is needed to use tcpcrypt-based authentication instead of SSL authentication.

Things are not quite so clean if application programmers manually create sockets using the BSD socket APIs instead of BIO, feeding them directly into `SSL_accept` and `SSL_connect`. These sockets will not have the necessary options set, and so tcpcrypt would disable itself even though the SSL library is capable. In such cases, if upgrading the application is not possible, then a `sysctl` could be used to set the application bit on by default on specific TCP ports.

Batch signing is implemented per SSL context. A single worker thread (per SSL context) waits on a semaphore for work and batch signs all session IDs it finds on its work queue. The signer thread then wakes up all threads corresponding to the session IDs signed. For batch signing to work, the SSL server must be multithreaded. We note that this implementation naturally scales depending on load: if a single client needs a signature, it is produced right away; when under load, multiple client session IDs will be batch signed to amortize cost. Our OpenSSL patch and batch signing code total 700 lines of code.

### 5.4 Password based authentication

We implemented the weak password authentication scheme in Section 4.2 as well as the strong scheme from Section 4.3. The weak scheme uses CMAC-AES as the MAC, and employs IBM's CMAC patch [21] for OpenSSL. We implemented the strong authentication scheme ourselves (500 lines of code) using OpenSSL's built-in support for NIST Prime-Curve P-256.

## 6 Performance and compatibility

If we are to achieve our ultimate goal of encrypting almost all Internet traffic, then the cost of doing so must be sufficiently low that the cost/benefit trade-off makes sense, even when the benefits are small. What then are the costs of running tcpcrypt? Roughly, the performance cost breaks down as follows:

- The cost of the tcpcrypt key exchange.
- The cost of encrypting and MACing every packet on the wire.
- The cost of authentication over tcpcrypt, for applications that choose to authenticate.

We must demonstrate that the first two are small enough they will not significantly degrade the performance of the vast majority of servers (clients are rarely the bottleneck, as they handle only a few connections per second at most). We must also demonstrate that the third is at least as cheap as current deployed solutions.

In addition, we must also demonstrate compatibility. Tcpcrypt must not cause connections to fail that would succeed without tcpcrypt.

### 6.1 Connection setup rate

Just how fast do servers need to accept connections in practice? It is hard to get firm numbers. *YouTube* gets 1 billion hits per day [12], thus averaging about 11,500 hits per second. *Facebook* currently gets about 260 billion page views per month [20], or around 100,000 per second. Of course a page may require more than one TCP connection, but with HTTP/1.1 the number will be fairly small. Facebook also has over 30,000 servers [24]. Not all these are front-end servers, but even so it becomes clear that the number of connections that need to be handled per second on each server is unlikely to be more than a few thousand.

To get another perspective, we can examine what an untuned operating system running an untuned web server can achieve. This tells us how default configurations perform, and so what a typical server administrator might expect. Our test machines are eight-core (two Intel Xeon X5355 CPUs) running Linux 2.6.32. Each has 13 1Gb/s NICs connected to client hosts via a LAN. Multiple clients and parallel connections are needed to saturate the server. Untuned, these servers can handle 35,500 TCP connections per second in a simple connection setup and teardown test, or 28,400 connections per second running Apache serving a small static file.

To determine the effect of tcpcrypt, first we need a better control experiment because the untuned numbers above, although typical of most real-world installations, fail to fully utilize the machine, leaving some idle time. It took considerable tuning[4] to get the connection setup and teardown test to saturate all the cores. Such a setup is not realistic for normal operation, but we wish to compare against the best-case vanilla TCP, not one that leaves unused CPU cycles. We will compare this optimized TCP against SSL and tcpcrypt.

We expect tcpcrypt to slow down TCP's connection throughput in two main ways. First, uncached tcpcrypt connections use public key operations to setup a connection. This cost is predominantly born by clients, which perform the more expensive RSA decryption operation. We use 2048-bit RSA-3 keys in all benchmarks.

Second, packets are MACed and thus require more CPU cycles and memory accesses. Even with connection caching, which avoids the need for public key cipher operations, four out of six of the packets in an `accept/close` cycle are MACed (two ACKs and two FINs). We therefore expect a performance degradation both in the uncached and cached connection cases, though uncached connections will be more expensive.

We expect SSL to perform less well than tcpcrypt for two reasons. First, it requires more RTTs to complete a connection because SSL's handshake can only start after TCP's handshake. More notably, uncached SSL connections should be much slower than tcpcrypt's because an SSL server performs the more expensive RSA decryption operation. However SSL also authenticates the server, so this is not an apples-to-apples comparison. We shall examine the cost of tcpcrypt's authentication in Section 6.2.

| Protocol | Connection rate (conn/s) Native | Divert |
|---|---|---|
| TCP server | 98,434 | 61,515 |
| tcpcrypt server (cached) | 70,044 | 38,832 |
| tcpcrypt server (uncached) | 27,070 | 21,908 |
| SSL server (cached) | 39,785 | 27,348 |
| SSL server (uncached) | 754 | 743 |
| tcpcrypt client (uncached) | 794 | 749 |

Table 1: Connection setup rate of tcpcrypt.

Table 1 shows the results. Both the cached case (same client reconnecting) and uncached case (new client, requiring public key cipher operations) are shown. The two columns benchmark our two tcpcrypt implementations: the kernel one ("Native") and the userspace divert socket one. To get divert numbers for TCP and SSL, we divert all traffic to userspace and back to the kernel; although this isn't useful, it allows us to separate out the different costs and see the overhead of the divert socket separately from additional protocol mechanism in the tcpcrypt userspace implementation.

Tcpcrypt outperforms SSL in the uncached case by a large margin due to reversing the asymmetric RSA costs; the client bears this cost. Tcpcrypt's cached performance is also better than SSL. We note that our kernel implementation is not fully optimized, so it may well be possible to get even greater performance. For example, we could encrypt and MAC data while copying it from userspace rather than doing it on a later pass. This would be an optimization similar to that for checksum calculation already used in Linux.

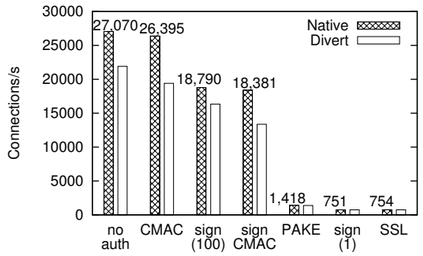
While TCP can be up to 41% faster for cached connections and 3.6× faster for uncached ones, we believe that the absolute performance numbers of tcpcrypt have their own merit. Recall that a heavily loaded website like Youtube averages 11,500 connections per second and tcpcrypt should be able to sustain such high load. Also recall that our untuned default configuration server can only handle 35,500 connections running the same benchmark of Table 1, also a target that tcpcrypt can meet if some sessions are cached.

The divert socket implementation is slower than our kernel one due to the multiple copies needed for each packet, from the kernel to userspace, and then back to the kernel. Furthermore the userspace implementation needs to (wastefully) duplicate TCP functionality already present in the kernel such as checksum calculations and protocol control block lookups. However, we believe that the absolute performance numbers of our divert socket implementation are sufficient for many situations, especially on clients, where simple installation may be a priority over performance.

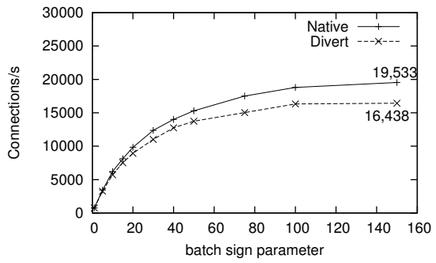
### 6.2 Authenticated connection setup rate

While Table 1 included SSL as a reference point, it cannot be used to directly compare the two systems because SSL performs authentication by default and thus is stronger than unauthenticated tcpcrypt. As tcpcrypt leaves authentication to applications, we are free to examine different authentication schemes. Our authentication benchmarks cover: tcpcrypt with batch signing (SSL replacement), CMAC password-based mutual authentication (vulnerable to offline dictionary attacks), batch signing combined with CMAC, and PAKE$_2^+$ password-based mutual authentication (both resistant to offline dictionary attacks). The benchmark and setup is identical to our previous benchmark, but with added application-level authentication after connection setup. We expect tcpcrypt with batch signing to outperform SSL when

[4] This involved running multiple instances of the benchmark on different ports to avoid kernel locks on `accept`. We set the affinity of each benchmark to one CPU, and used a different NIC per benchmark, with the NIC's interrupt affinity set to the same CPU as the benchmark using the NIC. This resolved in optimal packet scheduling and load balancing that finally brought the system to zero idle time.




(a) Authentication performance comparison.




(b) Batch signing.

Figure 3: tcpcrypt's authenticated connection setup rate.

batching more than one request, as RSA signatures will be amortized. We expect CMAC to outperform RSA-based authentication, because it uses symmetric cryptography only. Our PAKE implementation is so far unoptimized, but even so we expect it to be faster than RSA because it replaces the expensive RSA signature with a few elliptic-curve operations.

Figure 3 shows tcpcrypt's authenticated connection setup rate when using our kernel implementation ("Native") and our userspace divert socket one. Batch signing performs differently depending on the size of the batch and Figure 3(b) shows how this scales. Most of the benefits of batch signing arise even with a parameter as low as 100, a number of concurrent clients easily reached when the server is under load. Figure 3(a) clearly shows that there is a range of performance characteristics which applications may choose from. With SSL instead, applications are forced to use relatively low performance one-way authentication. Clearly, one size does not fit all. With tcpcrypt, applications can choose any combination of one-way or two-way authentication and higher performance at lower security or lower performance at higher security. For example, a busy web forum might choose CMAC for its authentication as it requires two-way authentication and high performance, but perhaps is not so security-critical that it needs to thwart offline dictionary attacks. This setup would perform 36x faster than SSL on uncached connections, providing stronger (two-way) authentication. A bank instead, might choose PAKE for its authentication, performing slower, but still twice as fast as SSL. Alternatively if a certificate is available, signing plus CMAC could be 24x faster than SSL and still resist offline dictionary attacks. A site requiring only one-way authentication, like a checkout from an online shop that does not require login, can perform up to 26x faster than SSL when loaded and handling over 150 concurrent requests. Tcpcrypt with batch signing is therefore a viable drop-in replacement for SSL, as in all cases its connection setup performance is superior (we shall examine data throughput in Section 6.4). Authentication adds little cost to tcpcrypt: 2% penalty with CMAC or 28% with batch signing under load. We believe this performance to be practical for many servers.

| Protocol | Connect time (ms) LAN | WAN |
|---|---|---|
| TCP | 0.2 | 105 |
| Tcpcrypt cached | 0.3 | 103 |
| Tcpcrypt not cached | 11.3 | 219 |
| Tcpcrypt CMAC | 11.4 | 320 |
| Tcpcrypt PAKE | 15.2 | 426 |
| SSL cached | 0.7 | 210 |
| SSL not cached | 11.6 | 321 |

Table 2: Connection setup time.

For most clients the performance of the divert socket implementation will be sufficient, providing an easily installed alternative.

Hardware is often used to offload expensive public key cryptography. For example, Sun's UltraSPARC T1 has a Modular Arithmetic Unit for RSA, and can do 2,300 2048-bit signatures per second using all 32 cores [18]. Tcpcrypt outperforms this using only eight general purpose cores, showing how careful protocol design can avoid the need to throw hardware (and money) at the problem. We argue that offloading asymmetric encryption is no longer needed for network encryption.

### 6.3 Connection latency

Throughput is not the only important metric—connection setup latency is also important. We compare the connection setup time from the client's point of view for TCP, SSL and tcpcrypt. We expect tcpcrypt to setup connections faster than SSL because tcpcrypt's handshake requires fewer round trips. Table 2 shows the time to establish a connection on a LAN (0.2ms RTT) and on a WAN (100ms RTT).

When the connection is cached, tcpcrypt adds very little delay to TCP because no extra RTTs are needed. Tcpcrypt does extra work to advance keys and MAC the ACK, hence it takes fractionally longer. SSL cached takes considerably longer because its negotiation can only start after TCP's handshake finishes whereas tcpcrypt uses the three-way handshake. In the non-cached case tcpcrypt and SSL perform similarly on the LAN as RSA dominates the cost. The main difference is that tcpcrypt is client-limited whereas SSL is server-limited. On the WAN, RTT dominates; tcpcrypt costs one RTT more than TCP, but one RTT less than SSL as it needs fewer messages to complete the handshake. Authenticating an uncached tcpcrypt connection, for example using CMAC or PAKE, adds extra latency.

With batch-signing there might be a concern that the queuing of requests to be signed might add extra latency. In fact this is not the case—our implementation signs whatever queue is available as fast as it can. Even the fact that tcpcrypt with signing requires two RSA operations does not add to latency—the expensive decrypt operation on the client takes place in parallel with the sign operation on the server, so negligible extra latency is required beyond the extra RTT needed for authentication.

The main effect of batch signing is in fact to reduce latency as the server becomes loaded. This is shown in Figure 4, which graphs connection latency against the number of connections per second the server handles. As the load increases eventually the server saturates and the latency increases extremely rapidly. The figure shows SSL latency and tcpcrypt latency when the maximum batch size has been artificially limited to 1, 5 and 10. SSL and tcpcrypt with a batch size of one are indistinguishable on this graph, so we only plot one line. It is clear that when the server has CPU cycles to spare, the batch size has no adverse effect on latency. In fact, quite the reverse—batching reduces the variance (the plot shows 10$^{th}$ and 90$^{th}$ percentiles as error bars), because short-term variations in arrival rate map into variation in batch size rather than variation in CPU load. More importantly, allowing larger batch sizes allows the server to saturate much later, and so maintain this low latency across a much wider range of server workloads.

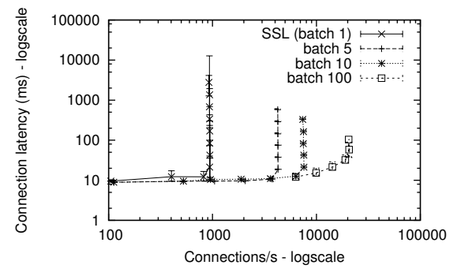



Figure 4: Latency as connection rate increases.

### 6.4 Data transfer rates

We now account for the cost of symmetric encryption and determine the maximum data throughput one can expect with tcpcrypt. We benchmark data throughput when transmitted with TCP, tcpcrypt and SSL. To fully saturate the CPU we ran one benchmark program per core and NIC pair, setting the affinity of the benchmark and NIC to a particular core. Otherwise, packet scheduling was suboptimal resulting in idle time. We expect SSL and tcpcrypt performance to be similar as both are doing AES128 and HMAC-SHA1. Obviously, vanilla TCP will be fastest as it need not encrypt or MAC.

| Protocol | Transfer Throughput (Mbit/s) Native | Divert |
|---|---|---|
| TCP | 12,954+ | 3,357 |
| tcpcrypt AES-SHA1 | 3,968 | 1,752 |
| SSL AES-SHA1 | 3,692 | 1,939 |

Table 3: tcpcrypt's data throughput.

Table 3 shows the data throughput of tcpcrypt, for our kernel implementation ("Native") and our userspace divert socket one. We were unable to saturate the CPU on the TCP benchmark (11% idle time) as we saturated all available NICs on the server. Tcpcrypt outperforms SSL by 7.4%. This was unexpected as the two essentially perform the same tasks: AES and SHA1. We are using different implementations for AES (Linux's kernel *vs.* OpenSSL) though we found the two to perform similarly when benchmarked individually. The fundamental differences between tcpcrypt and SSL are that SSL must do its own data segmentation and encapsulation (in addition to TCP's) thus needs more work than tcpcrypt. SSL MACs at a message boundary which can span multiple packets, whereas tcpcrypt must MAC once per packet. Tcpcrypt is MACing slightly more data as it includes packet headers, though the cost of SSL's message encapsulation seems to outweigh the additional bytes MACed by tcpcrypt. Overall, however, CPUs are powerful enough to fully encrypt a one Gigabit link, and in fact even more. Client machines seldom have more capacity than that, and even our userspace implementation provides sufficient performance for those cases.

Most relevant to servers, higher rates are possible by using faster ciphers and MACs; tcpcrypt achieves 7,486Mbit/s using Salsa20/12 and UMAC. High-speed AES is possible too now that AES-enhanced CPUs are becoming ubiquitous, like Intel's Westmere CPU [8], Sun's UltraSPARC T2 [2] and VIA's processors [1]. On a dual-core 3.33GHz desktop i5 with a 10Gb/s NIC,

| Protocol | Apache, static content (req/s) Native | Divert |
|---|---|---|
| TCP | 60,156 | 27,196 |
| tcpcrypt (cached) | 42,440 | 20,034 |
| tcpcrypt (uncached) | 19,153 | 14,215 |
| SSL (cached) | 19,787 | 12,063 |
| SSL (uncached) | 737 | 705 |

Table 4: Apache performance serving static content.

tcpcrypt performed 8,835Mbit/s using AES-UMAC, even without TCP segmentation offloading and optimizations in tcpcrypt. As an experiment, we were able to saturate 10Gb/s by using jumbo frames or by overclocking the box to 3.75GHz. We thus soon expect CPUs that will permit 10Gig AES networking—in fact, this is likely possible today if a six-core server i5 is used.

### 6.5 Application performance: Apache

We now study the overhead of tcpcrypt when used in a real application. We test the Apache web server (v2.2.11) serving a 44 byte static file. This setup has low application overhead, emphasizing overhead imposed by the networking stack. With a default configuration, our server can handle 28,400 requests per second though the CPUs remain unsaturated. To fully saturate CPUs, we must run multiple Apache instances, each on a different TCP port, serving traffic on a different NIC. Based on our microbenchmarks, we expect tcpcrypt to outperform SSL and have lower performance than TCP. We do not perform any authentication on this tcpcrypt benchmark, so SSL provides stronger guarantees in this case. However, as discussed earlier, authentication can be added to tcpcrypt at a relatively low cost if needed.

Table 4 shows the results of our Apache benchmark. Because real-world web traffic is a mix of new and returning clients, connection setup can quickly become a bottleneck for SSL. Tcpcrypt, on the other hand, maintains a high connection rate (31% of native TCP) even for new clients. Note also that the case of small, static files is a worst-case benchmark for connection setup. We tried benchmarking WordPress, a more CPU-intensive application. Neither tcpcrypt nor SSL caused a measurable slowdown. This test demonstrates that ubiquitous encryption is feasible when the application is the bottleneck, and in most cases even if it is not.

### 6.6 Compatibility

Incremental deployment is one of our chief goals. Essentially this entails gracefully falling back to TCP so that connections are guaranteed to succeed. Users will not enable tcpcrypt if doing causes their connections to fail. Tcpcrypt falls back gracefully so long as packets with the CRYPT option do not get dropped. Otherwise, tcpcrypt might indefinitely send SYN packets with the CRYPT option, and the connection would fail when it would succeed using a virgin SYN packet. To gauge whether this is a problem, we initiated tcpcrypt connections to the top 10,000 sites listed on Alexa. Specifically, we sent a SYN with the CRYPT-HELLO option set, expecting to get a SYN-ACK back. If not, we considered the packet dropped. We retransmitted SYNs to detect packet loss. This gives a rough estimate of how many connections would fail because of tcpcrypt.

Of the Alexa top 10,000 sites, we found 15 (0.015%) that did not respond with a SYN-ACK to a tcpcrypt SYN. Of these, three were on the same network. Given such a low failure rate, we are optimistic that tcpcrypt will work most of the time and can be safely deployed. However, by default, tcpcrypt will try to revert to standard TCP in case it does not receive a SYN-ACK after sending a few tcpcrypt SYNs to ensure reachability.

We do not expect tcpcrypt to suffer ECN's fate in terms of compatibility. ECN used reserved bits in the TCP header which would trigger IDSs and cause undefined behavior. Instead, tcpcrypt uses options as dictated by TCP's specification and is not anomalous in any way—for instance, even during re-keying the protocol design ensures that retransmissions always produce the same payload bytes for a given range of sequence numbers. We thus believe that tcpcrypt can safely be deployed on today's Internet as it will, for the majority of users, provide stronger security without breaking connections or noticeably reducing performance.

## 7 Related work

We categorize related work based on the networking stack layer it operates in. The network layer is dominated by IPSec-based solutions. IPSec [16] encrypts all data above the network layer. However, IPSec has not enjoyed widespread deployment and use, so a reasonable fear is that tcpcrypt could endure the same fate. Fortunately, several factors make it easier to deploy tcpcrypt and provide greater incentive to do so, leaving us some hope that ubiquitous encryption can succeed at the transport layer even if it has not at the network layer.

A big challenge to IPSec is that it breaks middleboxes that require access to the transport layer. Given the increasing prevalence of NAT in particular, this excludes a large portion of the population from using IPSec. Tcpcrypt, by contrast, operates at the transport layer and so avoids these problems. Another challenge for IPSec is that it is hard to create a notion of a "session" in a connection-less environment (the network layer). Thus, while IPSec is good at authenticating hosts to one another for purposes such as virtual private networks, it would be difficult and cumbersome to authenticate individual users, processes, and connections between hosts. Moreover, some transport-level security issues, such as protecting against wrapped acknowledgment numbers, are harder to reason about in IPSec.

Conversely, there are several incentives for deploying tcpcrypt that have no analogue with IPSec. One is that it can be integrated in a backwards-compatible way with SSL and significantly increase performance. By contrast, SSL over IPSec would require double-encryption, reducing performance. Second, TCP multipath requires a means of authenticating the same endpoint with multiple IP addresses, which tcpcrypt makes much easier. That said, tcpcrypt is less general than IPSec, which encrypts everything above IP, including UDP.

Better Than Nothing Security (BTNS) [26] is IPSec without a PKI, thus providing no security guarantees against active attackers. This is similar to default tcpcrypt. However, tcpcrypt additionally exposes the necessary hooks so that applications can perform authenticate in a variety of ways to guarantee security. Opportunistic encryption using IKE [23] specifies how to use IPSec with certificates obtained from DNSSEC. Tcpcrypt would need application support to integrate with DNSSEC.

We found no privacy solutions integrated into the transport layer. There are, however, integrity solutions. TCP MD5 [11] and AO [27] provide authentication and integrity protection within TCP. Tcpcrypt provides more functionality than these options by providing encryption. Moreover, tcpcrypt is fundamentally different as it requires no user setup. The session is established using ephemeral keys, and authentication can happen over the session itself. TCP MD5 and AO require establishing pre-shared secrets through out-of-band means. The main use of TCP MD5 and AO is to protect manually configured BGP sessions, which tcpcrypt can do as well by disabling unauthenticated RST packets. Also, TCP AO does not interoperate with NATs (which is okay for its intended use, as BGP is not usually spoken through NATs).

The dominant encryption solution above the transport layer is SSL [22]. Tcpcrypt offers a number of benefits over SSL, including better server performance, intrinsic forward secrecy, and integrity protection for the TCP session itself. Tcpcrypt is also more general, as it supports arbitrary authentication mechanisms and does not require a PKI. Finally, tcpcrypt is backward compatible with legacy applications and legacy hosts, which should ease ubiquitous deployment. Being more general, tcpcrypt can be used as a drop-in replacement for SSL, and we have in fact produced an SSL library that falls back to SSL if tcpcrypt is unavailable.

ObsTCP [17] also aims to provide opportunistic encryption, but is only designed to provide security in aggregate, not for specifically targeted connections. The author states, "We continue to advocate TLS as the only user facing transport security," meaning ObsTCP will duplicate encryption done by TLS, not protect transport headers, and not integrate with application-level authentication. ObsTCP requires no new TCP options and no extra round trips for connection setup, but the downside is that applications must be modified and that the first connection between two hosts remains unencrypted unless one knows that the other supports ObsTCP.

While tcpcrypt combines only well-known techniques, no other existing protocol can accomplish all of its goals. Specifically, tcpcrypt can be incrementally deployed on today's Internet, works out-of-the-box (even through NATs) without manual configuration, provides high enough performance to be on by default, and allows applications to integrate transport-layer security with arbitrary higher-level authentication techniques. The Internet demands higher security, hardware is ready for it, and the cryptographic techniques were waiting to be pieced together; tcpcrypt does so, and we believe our evaluation shows it could be readily deployed.

## 8 Conclusion

Tcpcrypt demonstrates that ubiquitous encryption of TCP traffic is technically feasible on modern hardware. By leveraging the asymmetry of common public key ciphers, it is possible for a server to accept and service around 20,000 tcpcrypt connections per second without session caching. Even higher rates are possible with caching. Data transfer rates are not an issue either; AES-SHA1 encryption and integrity protection can be done at several gigabits per second without hardware support on 2008-era hardware. The newest Intel CPUs incorporating AES instructions are even faster—tcpcrypt can reach 9Gb/s using AES-UMAC on a dual-core i5 desktop, suggesting that six-core i5 servers should handle 10Gb/s. These results suggest that tcpcrypt should have a negligible impact on the vast majority of applications.

The main contribution of this work is not performance, though this is a prerequisite. There are no new cryptographic primitives, nor is the protocol especially novel. The main contribution is from putting well-understood components together in the right way to permit rapid and universal deployment of opportunistic encryption, and then providing the right hooks to encourage innovation and deployment of much better and more appropriate application-level authentication. This ability to integrate transport-layer security with application-level authentication largely obviates the need for applications to encrypt their own network traffic, thereby minimizing duplication of functionality.

As an example, we showed how a simple batch-signing server-authentication scheme can leverage tcpcrypt to provide forward secrecy and the same security as SSL while handling 25 times the connections per second. At the same time, the protocol allows an SSL server to fall back gracefully to regular SSL behavior when one or the other side cannot utilize tcpcrypt for authentication.

We also demonstrated the use of tcpcrypt to bootstrap both weak and strong password-based mutual authentication (using CMAC and PAKE respectively). Password-based authentication without mutual authentication, even over SSL, really should be a thing of the past. Using tcpcrypt with batch signing and CMAC mutual authentication is strictly stronger than HTTP Digest authentication over an SSL session, and more than 20 times faster. Using tcpcrypt and our unoptimized PAKE implementation is almost twice as fast as SSL, and provides stronger security. Many other authentication mechanisms are possible; we believe that tcpcrypt's generality and simple application-level hooks are exactly what is required to get application writers to think about the form of authentication they really need, once they can address authentication separately from the question of how to encrypt session data.

Finally, tcpcrypt interoperates seamlessly with legacy applications, TCP stacks, and middleboxes, making it easy to deploy incrementally. For all of the above reasons, we believe that it now makes sense to make transport-layer encryption the default. Make it happen by installing tcpcrypt from `http://tcpcrypt.org`.

## Acknowledgments

We thank Alan Eustace, Daniel Giffin, Eric Grosse, Brad Karp, Adam Langley, the anonymous reviewers, and our shepherd, Nikita Borisov, for information, suggestions, feedback, and other assistance. This work was funded by gifts from Intel (to Brad Karp) and from Google, by NSF awards CNS-0716806 (A Clean-Slate Infrastructure for Information Flow Control) and CCR-0331542 (PORTIA), and by the EU FP7 Trilogy project.

## References

[1] VIA Padlock Security Engine.

[2] Ultra-FAST Cryptography on the Sun UltraSPARC T2. `http://blogs.sun.com/bmseer/entry/ultra_fast_cryptography_on_the`.

[3] BITTAU, A., HANDLEY, M., AND LACKEY, J. The final nail in WEP's coffin. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 386–400.

[4] BONEH, D., AND SHOUP, V. A graduate course in applied cryptography. Version 0.1, from `http://cryptobook.net`, 2008.

[5] FEDERAL COMMUNICATIONS COMMISSION. Commission orders Comcast to end discriminatory network management practices. `http://hraunfoss.fcc.gov/edocs_public/attachmatch/DOC-284286A1.pdf`.

[6] FRANKS, J., HALLAM-BAKER, P., HOSTETLER, J., LAWRENCE, S., LEACH, P., LUOTONEN, A., AND STEWARD, L. HTTP authentication: Basic and digest access authentication. RFC 2617, 1999.

[7] GRANBOULAN, L. How to repair ESIGN. In *Security in Computer Networks* (2003), vol. 2576 of *LNCS*, pp. 234–240.

[8] GUERON, S. Intel Advanced Encryption Standard (AES) Instructions Set. Intel White Paper, Rev 03.

[9] HANDLEY, M., PAXSON, V., AND KREIBICH, C. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2001), USENIX Association, pp. 9–9.

[10] HANSTEEN, P. N. M. *The Book of PF - A No-Nonsense Guide to the OpenBSD Firewall*. No Starch Press, 2007.

[11] HEFFERNAN, A. Protection of BGP Sessions via the TCP MD5 Signature Option. RFC 2385, 1998.

[12] HURLEY, C. Y,000,000,000utube. The Official YouTube Blog, `http://youtube-global.blogspot.com/2009/10/y000000000utube.html`.

[13] JACOBSON, V., BRADEN, R., AND BORMAN, D. TCP extensions for high performance. RFC 1323, 1992.

[14] JONCHERAY, L. A simple active attack against tcp. In *SSYM'95: Proceedings of the 5th conference on USENIX UNIX Security Symposium* (Berkeley, CA, USA, 1995), USENIX Association.

[15] KATZ, J., AND LINDELL, A. Y. Aggregate message authentication codes. In *Topics in Cryptology – CT-RSA* (2008).

[16] KENT, S., AND ATKINSON, R. Security Architecture for the Internet Protocol. RFC 2401, 1998.

[17] LANGLEY, A. Obfuscated TCP. `http://code.google.com/p/obstcp/wiki/Transcript`.

[18] LIN, C.-C. RSA Performance of Sun Fire T2000. `http://blogs.sun.com/chichang1/entry/rsa_performance_of_sun_fire`.

[19] MATHIS, M., MAHDAVI, J., FLOYD, S., AND ROMANOW, A. TCP selective acknowledgement options. RFC 2018, 1996.

[20] MCCARTHY, C. Pingdom: Facebook is killing it on page views. CNET News, `http://news.cnet.com/8301-13577_3-10428394-36.html`.

[21] PETER WALTENBERG. AES-GCM, AES-CCM, CMAC updated for OpenSSL 1.0 beta 2 – revised.

[22] RESCORLA, E. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley Professional, 2000.

[23] RICHARDSON, M., AND REDELMEIER, D. Opportunistic Encryption using the Internet Key Exchange (IKE). RFC 4322 (Informational), December 2005.

[24] ROTHSCHILD, J. High performance at massive scale - lessons learned at Facebook. Seminar at UCSD.

[25] SCHILLACE, S. Default HTTP access for Gmail. `http://gmailblog.blogspot.com/2010/01/default-https-access-for-gmail.html`.

[26] TOUCH, J., BLACK, D., AND WANG, Y. Problem and Applicability Statement for Better-Than-Nothing Security (BTNS). RFC 5387 (Informational), November 2008.

[27] TOUCH, J., MANKIN, A., AND BONICA, R. The TCP authentication option. Internet draft (work in progress), July 2009.



# Automatic Generation of Remediation Procedures for Malware Infections

*Roberto Paleari*[1], *Lorenzo Martignoni*[2], *Emanuele Passerini*[1],
*Drew Davidson*[3], *Matt Fredrikson*[3], *Jon Giffin*[4], *Somesh Jha*[3]

[1]*Università degli Studi di Milano*
{roberto, ema}@security.dico.unimi.it

[2]*Università degli Studi di Udine*
lorenzo.martignoni@uniud.it

[3]*University of Wisconsin*
{davidson, mfredrik, jha}@cs.wisc.edu

[4]*Georgia Institute of Technology*
giffin@cc.gatech.edu

## Abstract

Despite the widespread deployment of malware-detection software, in many situations it is difficult to preemptively block a malicious program from infecting a system. Rather, signatures for detection are usually available only after malware have started to infect a large group of systems. Ideally, infected systems should be reinstalled from scratch. However, due to the high cost of reinstallation, users may prefer to rely on the remediation capabilities of malware detectors to revert the effects of an infection. Unfortunately, current malware detectors perform this task poorly, leaving users' systems in an unsafe or unstable state. This paper presents an architecture to automatically generate *remediation procedures* from malicious programs—procedures that can be used to remediate all and only the effects of the malware's execution in any infected system. We have implemented a prototype of this architecture and used it to generate remediation procedures for a corpus of more than 200 malware binaries. Our evaluation demonstrates that the algorithm outperforms the remediation capabilities of top-rated commercial malware detectors.

## 1 Introduction

One of the most pressing problems faced by the Internet community today is the widespread diffusion of malware. To defend against malware, users rely on signature- or behavior-based anti-malware software that attempts to detect and prevent malware from damaging an end-host. Unfortunately, in many cases detection and prevention are not possible. Malware authors have perfected the practice of automatically creating a large number of *variants*, or malware that appears new to detectors but exhibits the same behavior when executed. For new malware and variants, signatures for detection are rarely available by the time malware reaches a network, leaving a time window in which systems are susceptible to infection. In these situations, the ability to detect and remove the malware after infection is not enough—it is also imperative that any harmful changes to the system made by the malware are *remediated* (or reverted).

The safest way to remediate a system is to format the permanent storage and re-install the operating system from scratch. While effective, this approach is also costly and usually results in a loss of valuable personal data, particularly when data backups are incomplete or non-existent. Rather, end-users and administrators may prefer to remove only those resources left behind by the malware, leaving the rest of the system intact. Unfortunately, current anti-malware products perform poorly at this task. A recent study demonstrated that even top-rated commercial anti-malware software fails to revert the effects of all the actions performed by malware during infections [15]. Needless to say, partially-remediated systems are unstable and prone to error.

In this paper, we present a system that automatically generates *remediation procedures* from malware binaries. These remediation procedures can be executed on infected systems to restore the state to a clean configuration, and are capable of remediating the effects of a malware sample *a posteriori*, without observing the infection take place. The fact that our remediation procedures are generated to cover a particular malware binary, rather than a specific sequence of system events resulting in an infected state [7, 19], amounts to a substantial break from previous technologies. Using our system, one can generate a single general-purpose executable that is capable of reversing the effects of a malware sample on an arbitrary number of hosts *after the fact*. In other words, one does not need to be aware of our system, or make use of it, until *after* the infection takes place. To achieve this goal, we rely on a combination of dynamic program analysis and semantic generalization to produce models of infection behavior that are resilient to common malware anti-analysis techniques, such as the use of nondeterminstic file names or the omission of malicious behavior on some runs of the program. Then, we translate these

behavior models directly into executable procedures that remediate the effects of a malware infection.

We have implemented our ideas in a prototype tool. Using the prototype, we automatically generated remediation procedures on a corpus of more than 200 binary malware samples belonging to approximately 50 distinct families. We evaluated the practical effectiveness of each procedure by testing its ability to recognize all of the harmful effects of a malware execution (*true positives*) while leaving benign aspects of the system intact (*true negatives*). The results of our evaluation attest to the effectiveness of our technique: *in total, we reversed 98% of the harmful effects while generating only a single false positive*, although we were not able to remediate user-specific resource changes such as deleted documents and personal file mutations. In contrast, the best commercial anti-malware product remediated only 82% of the effects of our corpus.

In summary, we make the following contributions:

- We present an architecture to automatically generate remediation procedures given binary malware samples. To the best of our knowledge, our architecture is the first to work under the assumption that information relating to a specific infection is not available; rather, characteristic infection patterns are observed and *generalized* to produce effective procedures in this setting.
- We evaluated an implementation of our framework on on more than 200 real malware samples and found that it was able to remediate the resulting infections more effectively than existing commercial antivirus products. We have made this implementation available as an open-source package [1].

The rest of this paper is organized as follows: In Section 2 we discuss related work. In Section 3.1, we describe the problem that our architecture solves by presenting a realistic example, and in Section 3.2 we outline our approach, relating it to the example. In Section 4, we formalize the problem of malware remediation and present the technical details of our approach. In Section 5, we evaluate the effectiveness of our approach by testing a prototype implementation against real malware. In Section 6 we discuss the limitations of our approach, the security implications, and potential avenues for future work. We present concluding remarks in Section 7.

## 2 Related Work

Our contributions relate to ongoing research on behavior-based malware analysis, on the execution of untrusted applications in trusted systems, and on the automatic generation of signatures to detect malicious network traffic.

**Behavior-Based Malware Analysis:** The prevalence of packed, polymorphic, and metamorphic malware highlights the deficiencies of traditional detection approaches based on syntactic signatures. This has urged researchers and security practitioners to focus on solutions that base policy on the behavior exhibited by untrusted software. Behavior-based techniques attempt to infer security-relevant information about an untrusted program either by analyzing it statically [16] or by observing its operation dynamically [1, 11, 21]. The major drawback of current behavior-based techniques is their high computational overhead. Recently, Kolbitsch et al. developed an efficient analysis solution intended to replace traditional anti-malware on the desktop [8]. Closer in spirit to the work presented here is that of Christodorescu et al. [4]. They described an automatic approach that derives formal specifications of malicious behavior by comparing the observed dynamic behavior of malicious and benign applications. Their technique uses dependence graphs, which express the relationships among various low-level behavior events, and is similar in many ways to our high-level behavior abstraction component (see Section 4.2.1). Another area of much recent activity is that of automatic classification of malware into families [2, 18]. For that type of work, malware is grouped into clusters, which correspond to families, by some notion of behavioral similarity. Our technique uses a form of behavioral grouping as a means to remediate a system, but we go further than malware classification by attempting to remove the harmful effects of the malware on the system.

**Execution of Untrusted Applications:** In addition to work that attempts to detect or prevent the execution of malicious software, some work has been done to mitigate the harmful effects of software *a posteriori*. Hsu et al. presented a framework for automatically repairing an infected system after monitoring the execution of the malware [7]. The actual work of remediating a system given a detailed description of the malicious execution is similar to the way that we construct remediation procedures from generalized behavior models. Liang et al. described an alternative approach called Alcatraz [10]. In Alcatraz, an untrusted application is executed inside of a sandbox, and any change it makes is not committed until the program is confirmed to be innocuous. The manner in which a program is deemed innocuous is considered orthogonal to the main issue of sandboxing. The idea was later tweaked [19] so that all state changes made by an application are *cached*, and upon program termination the user decides whether or not to keep any changes. The primary differences between these techniques and the one presented in this paper is that they rely on information regarding specific execution traces, whereas our remediation procedures use generalized notions of the behavior of a malware instance. As such, our system can remediate harmful effects of malware, including some effects that were not observed in a trace.

**Automatic Signature Generation:** The generalized behavior models that we use to construct executable remediation procedures can be viewed as generic signatures relating the effects of a malicious program on system resources. Different approaches have been proposed for automatically generating attack signatures. Polygraph [14] is one of the first systems proposed by researchers to address the problem of generating network signatures to detect polymorphic worms. Polygraph identifies invariant fragments of packets that are found in all the network flows generated by the same worm, since they are necessary for the worm to successfully exploit a given vulnerability. These fragments are then combined into signatures using different techniques. Hamsa [9] addresses the same problem using a different algorithm that identifies and combines invariants. Hamsa's signatures have better accuracy and are more resilient against attacks than Polygraph. Finally, Nemean [20] generates semantics-aware signatures to detect network intrusions. Nemean's methodology, consisting of high-level network traffic abstraction, clustering, and generalization using automata learning, is similar to ours. However, we operate on a fundamentally different domain than Nemean, which generates signatures of network packet traces.

## 3 Overview

In this section, we motivate our work using a realistic example of a malware infection and present our architecture by walking through the steps that it takes to remediate the example.

### 3.1 Motivation

Consider the malware whose pseudo-code is shown in Figure 1. This program generates a random filename located in the system directory, drops a malicious payload into the file, creates a new registry value that causes the payload to be executed at system boot time, tampers with the system's network name resolver (`c:\...\etc\hosts`), and infects a benign system library (`c:\windows\user32.dll`). Our goal is to generate a procedure that remediates infections caused by any possible execution of this code. In this case, recovery includes: (1) deleting the file containing a copy of the malicious payload, (2) deleting the registry key created to start the malware at boot, (3) disinfecting `c:\windows\user32.dll`, and (4) restoring the original configuration of the name resolver `c:\...\etc\hosts`. It is important that the effects of *all* malicious actions taken by the malware are removed. For example, consider what happens when (1), (2), and (3) are remediated, but not (4). In this case, all internet traffic on the host remains subject to hijacking by the malware, so the system is still in a dangerous configuration. Many commercial products would leave the system in this configuration [15].

Completely remediating the effects of the malware in Figure 1 is not as straightforward as the example might suggest. First, high-level source code is usually not available when dealing with real malware. Given the well-known difficulty of statically analysing adversarial binary code [13], this means we must partially rely on dynamic information. Although this example does not illustrate it, there is a possibility that the malware contains paths that are rarely executed under normal circumstances. Any harmful effects produced on such a path would be difficult to account for in a remediation procedure, because the problem of discovering such an effect dynamically is extremely difficult. Secondly, malware can appear to be nondeterministic by relying on subtle details in its environment, such as the system clock or pseudorandom number generator. This behavior is often present even on common paths, and is apparent in our example, despite its simplicity: Both the filename of the malicious payload and the name of the registry value used to activate the payload depend on randomness.

Given the limited nature of dynamic program information, it may be hard to generate a remediation procedure that precisely accounts for all of the nondeterminism in a program. Procedures that do so may mistakenly identify benign system resources as malicious and attempt to remediate them. Consider a remediation procedure that attempts to account for the nondeterminism in our example by looking for all files in the system directory with the suffix `.exe`. While this policy would effectively capture the nondeterminism in the payload filename, any attempt to remediate resources based on it would result in the unacceptable removal of benign executables. Conversely, procedures that do not attempt to generalize execution behavior are likely to miss some malicious effects that must be remediated. For example, after running the sample malware once, we might find that the payload is delivered in `c:\windows\poqwz.exe`. If a remediation procedure does not generalize this information and only ever looks for this file when remediating infections caused by other executions of this malware, then it will miss the payload file most of the time, as it is not

[1] The URL for this tool is `http://www.cs.wisc.edu/~mfredrik/remediate`

```
1  // generate random file and value names
2  filename = "po" + random_alpha() + random_alpha() + random_alpha() + ".exe";
3  valuename = (random_int() % 2) ? "qv" : "vq";
4  ...
5  // drop malicious code
6  f = CreateFile("c:\windows\" + filename, GENERIC_WRITE, ...);
7  WriteFile(f, malicious_buf, ...);
8  WriteFile(f, other_malicious_buf, ...);
9  ...
10 // start the newly created executable at boot
11 RegOpenKey(HKEY_LOCAL_MACHINE, "...\Windows\CurrentVersion\Run", &r);
12 if (RegQueryValueEx(r, valuename, NULL, REG_SZ, ...) == ERROR_FILE_NOT_FOUND)
13   RegSetKeyValue(r, valuename, REG_SZ, filename, ...);
14 ...
15 // infect user32.dll
16 g = CreateFile("c:\windows\user32.dll", FILE_APPEND_DATA, ..., OPEN_EXISTING, ...);
17 WriteFile(g, malicious_buf, ...);
18 ...
19 // hijack HTTP connections to www.google.com and www.citibank.com
20 h = CreateFile("c:\windows\system32\drivers\etc\hosts", ..., OPEN_EXISTING, ...);
21 ReadFile(h, buf, ...);
22 WriteFile(f, "67.42.10.3 www.google.com\n67.42.10.3 www.citibank.com", ...);
23 ...
24 // delete main executable
25 DeleteFile("c:\malware.exe");
```

Figure 1: Pseudo-code of a sample malicious program.

possible to observe the malware long enough to see all possible variants of the payload file name.

## 3.2 Architecture Overview

The architecture we have developed for generating remediation procedures from malware binaries is shown in Figure 2. It has three primary components: (1) an execution monitor that infers the malware's high-level behaviors from a low-level trace, (2) a component that *generalizes* the high-level behaviors from multiple executions of the malware, and (3) a component that produces executable remediation procedures from generalized behaviors. The entire system works sequentially, with each component using the information produced by the one preceding it.

**High-Level Behavior Extraction:** The high-level behavior extraction component (numbered 1 in Figure 2) analyzes the semantics of a program to produce a sequence of meaningful behaviors relevant to remediation. Because malware authors usually obfuscate their binaries, we rely on dynamic information to infer these behaviors; we execute binaries in a special environment (an emulator) to extract a low-level execution trace, perform analysis using manually constructed rules, and arrive at a high-level trace [11]. Table 1 lists the high-level behaviors we consider. Each behavior modifies the state of the system in some way and is parameterized by a set of arguments that determine which aspects of the system state are affected. The behaviors currently listed correspond to those that commonly occur in malware, that are mandatory to infect a system, and were constructed manually to reflect the salient behavioral features of most malware. However, our technique can be extended to operate over a wider set of high-level behaviors.

The environment in which a program runs typically affects its behavior, and malware often exhibits a certain degree of nondeterminism. To account for these factors, we collect several high-level behavior traces for each sample. To do so, we vary the environment by changing factors that malware typically rely on, such as locale, service pack level, and so forth. Although not supported by our current implementation, path exploration techniques [12] can be applied in this component to account for a more complete subset of the malware's behavior, as in Bouncer [5]. The lack of path exploration techniques is not a fundamental limitation of our system, and can be easily *plugged into* our system.

Our high-level behavior extractor would infer that the sample malware from Figure 1 demonstrates the `FileCreation`, `RegistryCreation`, `DropAndAutostart`, and `FileInfection` behaviors, with different arguments for `FileCreation`, `RegistryCreation`, and `DropAndAutostart` on each execution.

**Behavior Generalization:** After producing a set of high-level behavior traces for a malware sample, we attempt to account for nondeterminism by creating a general, abstract model of behavior that accounts for all of the concrete traces we observed (numbered 2 in Figure 2). Note that generalization attempts to *overapproximate* existing paths, thus encompassing future paths, rather than explore as many new paths as possible. In effect, this *patches* some of the incompleteness of dynamic



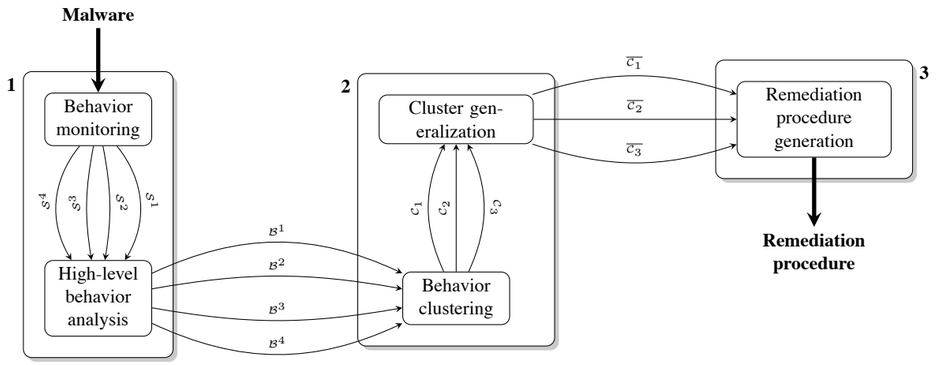



Figure 2: Architecture of the system for generating remediation procedures. In this figure, $\mathcal{S}$ denotes a system call trace, $\mathcal{B}$ denotes a high-level behavior trace inferred from a system call trace, $\mathcal{C}$ denotes a cluster, and $\bar{\mathcal{C}}$ denotes a *generalized* cluster.

analysis by extrapolating observed information to future, unseen executions of the malware. This is accomplished by recognizing when distinct behaviors from multiple high-level traces, with possibly different arguments, are actually instances of the same malicious activity. We refer to this matching of behaviors as *clustering*. When a cluster is identified, the arguments of its constituent behaviors are generalized to tolerate any differences that may be present in the actual values. Thus, nondeterminism is accounted for via overapproximation by ensuring that this generalization extends to future, unseen executions.

In the malware from Figure 1, our technique would cluster all instances of the same high-level behavior together. For example, all instances of `DropAndAutostart` would be clustered together and all instances of `FileInfection` would be clustered together. Because there is likely variation among the arguments of `DropAndAutostart`, we construct a regular expression to tolerate minor differences while ensuring that benign files are not mistakenly identified. The final result of the computation for this behavior would be a `DropAndAutoStart` behavior with generic file argument `c:\windows\po[[:alpha:]]{3}.exe` to generalize the random filename at line 2, generic registry key/value pair `...\CurrentVersion\Run` for the registry touched at line 11, and `(qv|vq)` for the registry value randomly created at line 3.

**Remediation Procedure Generation:** The third component of our architecture (numbered 3 in Figure 2) generates executable remediation procedures from the generalized behaviors produced in the previous step. The resulting procedure examines the state of the system on which it runs in search of symptoms of an infection, and removes the symptoms whenever possible. It attempts to match each resource (file, process, or registry key) on the system against the constraints associated with each generalized high-level behavior. For our running example, each file is matched against the regular expression `c:\windows\po[[:alpha:]]{3}.exe` associated with the first argument of the `DropAndAutoStart` behavior, another regular expression associated with the second argument, and a final one describing the content of the file. If such a file is found, then the registry values under the key `...\CurrentVersion\Run` are matched against the regular expression `(qv|vq)`. If such a value is found and its data matches the current filename being considered, then all of the resources (the file and registry key pair) are removed. Currently, we only produce remediation procedures that operate on system files. For technical reasons explained in Section 4, we do not handle user-specific files and resources. While this is a limitation of our current approach, we hope to remove it in future work.

# 4 Generating Remediation Procedures

In this section, we present the details of our system for generating remediation procedures. We begin by formalizing the problem solved by our system and continue component-by-component describing the algorithms used to solve the problem.

## 4.1 Problem Description

When malware runs on a system, it may infect the system by changing its persistent state in an undesirable way.

| | Behavior | Arguments | Description |
|---|---|---|---|
| **Resource creation** | `FileCreation` | File name and content | Creation of a new file |
| | `RegistryCreation` | Key name and content | Creation of a new registry value |
| | `DropAndAutostart` | File name and content. Key name and content | Creation of a new file and of a registry value containing its name (to execute the file automatically at every boot) |
| | `DropAndExecute` | File and process name | Creation and execution of a new executable |
| **Resource infection** | `FileInfection` | File name and content. List of preserved regions | Infection of an existing file |
| | `RegistryInfection` | Key name and content | Replacement of an existing registry value |
| **Resource deletion** | `FileDeletion` | File name | Deletion of an existing file |
| | `RegistryDeletion` | Key name | Deletion of an existing registry value |

Table 1: High-level behaviors considered for remediation.

For our purposes, the state $S$ of a system is modeled as an association from resource names $N$ to data from a domain $D$. Individual elements of $S$ are referred to as *resources*. To simplify notation, we let $\mathcal{S}$ stand for the set of possible system states. Because most malware is written for Windows platforms, our targeted resource namespace consists of Windows filenames, registry key and value names, and process names. The data domain is the set of all finite-length bit strings.

The infection behavior of a malware can be understood as a transition relation between system states. There are three ways in which the malware can modify the state of a system: *(1)* resources may be completely removed from the system, *(2)* new resources may be added to the system, and *(3)* the data corresponding to existing resources may be mutated. Because the infection behavior of a malware can be succinctly described in terms of these three operations and the resources over which they operate, we represent it using an *infection relation* $R \subseteq \mathcal{S} \times N \times \mathcal{S} \times \mathcal{S}$ that encodes this information. Intuitively, the infection relation describes the way in which a particular malware changes the state of a system. Given an element $(S, N_{rem}, S_{add}, S_{mut}) \in R$, the malware transforms state $S$ into a new state by removing the resources labeled by $N_{rem}$, adding the resources in $S_{add}$, and modifying the resources in $S_{mut}$. Note that the infection behavior is described as a relation rather than a function mapping. This is because of the fact that malware may behave nondeterministically when it infects a system—it may infect the same system state in different ways on two distinct executions.

After a given piece of malware has infected a system, the goal of remediation is to undo the effects of the infection, returning the system to a clean state. More precisely, given a malware binary, we seek to construct an infection relation for that malware that describes its behavior. We can then use the information in the infection relation to enact changes on the system that remediate the effects of the malware: restoring any files that were removed ($N_{rem}$) or mutated ($S_{mut}$), and removing files that were added ($S_{add}$). We package this functionality as an executable remediation procedure, as described in Section 3.2. In general, there are a number of approaches that may realize the goal of constructing the infection relation corresponding to a given malware. In this paper, we focus on applying dynamic analysis to the malware sample to extract the information necessary to construct the infection relation.

In practice, it is not usually possible to reconstruct the true infection relation from a malware binary. Rather, we compute a relation that *overapproximates* the actual behavior for a finite set of execution paths exhibited by the malware. For example, we overapproximate the resource names involved in the `DropAndAutoStart` behavior of Figure 1 by creating a regular expression that matches all of the resource names on the set of execution traces we observed. Furthermore, our approximate infection relations do not contain information regarding the removal or mutation of non-system files, as it is generally not possible to restore this state without additional information not encoded in the malware. Of course, using an approximate infection relation for remediation introduces the possibility of *false negatives* and *false positives*. A false negative occurs when the remediation fails to properly reverse the changes left by the malware. Similarly, a false positive occurs when remediation affects resources that were not touched by the malware. Both types of error are possible given the way we construct approximate infection relations. For example, false positives may result from the overapproximation of resource names with regular expressions, whereas false negatives may result from the fact that we do not account for all possible execution paths in the malware. Thus, it is our goal to construct an approximate infection relation that minimizes false positives and false negatives



## 4.2 System Details

This section details the specific algorithms and subsystems used in the three main components of our system (depicted in Figure 2).

### 4.2.1 High-Level Behavior Extraction

Intuitively, the problem of high-level behavior extraction is to derive a concise description of the behavior semantics demonstrated by a malware sample. Given a malware sample $m$ and a set $D$ of *high-level behavior templates* that describe events related to system state modification, the goal of this task is to produce a sequence of instances of the members of $D$, along with a corresponding low-level description of system events that match each template instance.

The set of behavior templates used in our prototype is given in Table 1. To infer high-level behaviors from a stream of system calls, we use *multilayer behavior specifications*, as proposed in previous work [11]. Although the details of the inference algorithm are beyond the scope of this paper, we give a brief account of the main points here. Each high-level behavior is described in terms of a hierarchical model. Each level of the hierarchy is composed of a set of *behavior summaries* and their accompanying *behavior graphs*. The graph for a given behavior summary encodes the behavior operationally, in terms of events and the dependencies among them. The events in a graph at a particular level are defined in terms of the summaries of levels lower in the hierarchy. The top level of the hierarchy corresponds to the final output of the inference, and the layers beneath it provide details of incremental specificity, until the lowest level is reached. In our prototype, the lowest level corresponds to a system call trace collected in a virtual environment. We use a modified version of QEMU [3] to monitor an application for its system call trace.

The nodes in the behavior graphs at each layer correspond to events that are observed by the monitor, and the edges correspond to data dependencies between the events. For example, in the graphs at the lowest level, system calls that operate on the same resource handle have edges between their representative nodes that reflect this dependency. At the highest level, this relationship is preserved by edges that denote the fact that the corresponding set of high-level behaviors operate on the same file. Representing high-level behavior graphs hierarchically has one crucial advantage: the same high-level behavior can be described in terms of multiple alternative intermediate behaviors. For example, our high-level behavior `DropAndAutostart` can be represented in terms of all possible low-level system call sequences that create a new file, write executable content into it, and then change the system configuration to activate the dropped file at boot time. Because there are numerous distinct ways to accomplish this high-level task in terms of system calls, it is important to account for all of them in a clean and straightforward way. Our heirarchical behavior model formalism allows this, and thus makes our system more resilient to this type of evasion.

Figure 3 shows a sample system call trace and two of the high-level behaviors extracted from it. The figure shows both the concrete graphs and the template instances that were matched. The first four system calls in the trace (members $s_1$ through $s_4$) are executed by the malware sample to replicate its payload into a new file. These calls are associated with the layer-1 behavior `FileCreation`. Similarly, the system calls $s_{11}, s_{13}$, and $s_{14}$ are associated with the layer-1 behavior `RegistryCreation`, and the last system call ($s_{41}$) with the behavior `FileDeletion`. Since the behaviors `FileCreation` and `RegistryCreation` are related, the algorithm infers the high-level layer-2 behavior `DropAndAutostart`, which represents the fact that the malware replicates and configures the system to execute the malicious payload at boot. Note that this high-level behavior was inferred hierarchically; the fact that `DropAndAutoStart` is present in the trace was inferred only from layer-1 behaviors, which were in turn inferred from system calls originally found in the trace. By modularizing the template definitions in this way, our high-level behavior inference technique gains a certain amount of resilience to obfuscations and differences in malware implementation [11].

### 4.2.2 Behavior Clustering

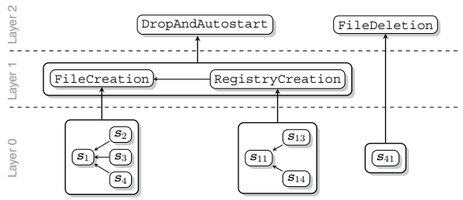
Given a set of high-level behavior traces $\{\mathcal{B}^1, \ldots, \mathcal{B}^m\}$ corresponding to multiple executions of the same malware sample, *behavior clustering* identifies elements of distinct traces that correspond to the same malicious activity. An *admissible clustering* for a given set of traces is a set of behavior sets $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ that satisfies two conditions:

1. All behaviors in a given cluster $\mathcal{C}_i$ have the same *type*. For example, all behaviors are of type `DropAndAutostart`.

2. The clustering *partitions* the set of all events in every execution trace: no behavior is in more than one cluster, and each behavior is in some cluster.

```
s1   NtCreateFile("poqwz.exe") → f
s2   NtWriteFile(f, "...malicious code...")
s3   NtWriteFile(f, "...other malicious code...")
s4   NtClose(f)
...
s11  NtOpenKey("Run")  →  r
s12  NtQueryValueKey(r, "vq") → FAILURE
s13  NtSetValueKey(r, "vq", "poqwz.exe")
s14  NtClose(r)
...
s21  NtOpenFile("...\system32\user32.dll")  →  g
s22  NtWriteFile(g, "...malicious data...")
s23  NtClose(g)
...
s31  NtOpenFile("c:\windows\hosts")  →  h
s32  NtReadFile(h, 1024) → "# Copyright (c)..."
s33  NtWriteFile(h, "67.42.10.3  www.google.com...")
s34  NtWriteFile(h, "67.42.10.3  www.citibank.com...")
s35  NtClose(h)
...
s41  NtDeleteFile("c:\malware.exe")
```

(a)




**High-Level Behavior Summaries**

```
DropAndAutostart("poqwz.exe", data, "Run", "vq",
                 "poqwz.exe")
FileCreation("poqwz.exe", data)
FileDeletion("c:\malware.exe")
RegistryCreation("Run", "vq", "poqwz.exe")
```

(b)

Figure 3: The system call trace for our sample `malware.exe` (a) and high-level behaviors generated from the trace (b).

In later stages of the system, it generalizes behaviors in the same cluster by overapproximating their argument values. Thus, desirable clusterings are those that lead to tighter overapproximations, while still grouping related behaviors together in order to allow generalization. As an example, Figure 4 shows two high-level traces of our sample malicious program. We denote the $j^{th}$ behavior observed in the $i^{th}$ execution trace as $b^i_j$. For these traces, we want to group behaviors $b^1_1$ and $b^2_1$ because they correspond to the same activity, and generalizing their arguments leads to a tight overapproximation: we can use regular expressions that match a fairly small set of strings (namely, `po[[: alpha :]]{3}.exe`). Similarly, we want to group $b^1_2$ with $b^2_2$ and $b^1_3$ with $b^2_3$. However, had the second trace contained another `DropAndAutostart` behavior for an executable named `avkiller.exe`, then clustering $b^1_1$ with this behavior would have resulted in a poor generalization. An optimal clustering is one that includes all related high-level behaviors so that generalization will create a powerful regular expression that finds all traces of a malicious behavior. On the other hand, an optimal clustering must not include unrelated high-level behaviors, as a generalization of such a cluster is likely to match benign system resources.

**Cluster Formation:** Exhaustively searching for the optimal clustering of $\{\mathcal{B}^1, \ldots, \mathcal{B}^m\}$ is infeasible, as there are an exponential number of possibilities. Thus, we do not attempt to find an optimal clustering and instead rely on the heuristic method shown in Algorithm 1. The algorithm begins by finding the execution trace with the greatest number of high-level behaviors $\mathcal{B}^{max}$, and creating an initial clustering by placing each $b^{max}_i$ in its own cluster $\mathcal{C}_i$. Then, for each remaining behavior trace $B^j$, the events are enumerated in execution order and added to the first cluster that satisfies the admissibility criterion discussed above. We discuss the details of matching event types below. If an event cannot be added to any existing cluster, then a new cluster is initialized with the current event. This process is repeated until no traces remain, at which point the current set of clusters is returned as the final result.

Intuitively, the heuristics in this algorithm rely on two asssumptions: *(1)* distinct executions of the malware exhibit similar malicious behaviors, and *(2)* the ordering of malicious behaviors between executions is similar. By selecting the trace with the greatest number of events to seed the clustering process and assuming that different executions contain a similar set of behaviors, we seek clusterings that group as many behaviors together as possible. By adding events to existing clusters in execution order and assuming that the order does not vary substantially between executions, we seek clusterings that match similar argument values, thus resulting in tighter overapproximations in the behavior generalization phase of the system. Furthermore, these heuristics allow our algorithm to operate efficiently: Algorithm 1 runs in time linear in the number of execution traces and the length of the traces.

For an example of how Algorithm 1 works, consider the two high-level execution traces depicted in Figure 4. As both traces are of equal length, the first is chosen, in this case $\mathcal{B}^1$. Clusters $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ are initialized with behaviors $b^1_1$, $b^1_2$, and $b^1_3$, respectively. $b^1_1$ and $b^2_1$ can then be matched, as they are both instances of the `DropAndAutostart` high-level behavior. Similarly, $b^1_2$ is matched to $b^2_2$, and $b^1_3$ is matched to $b^2_3$. Finally, the algorithm returns clusters $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ where



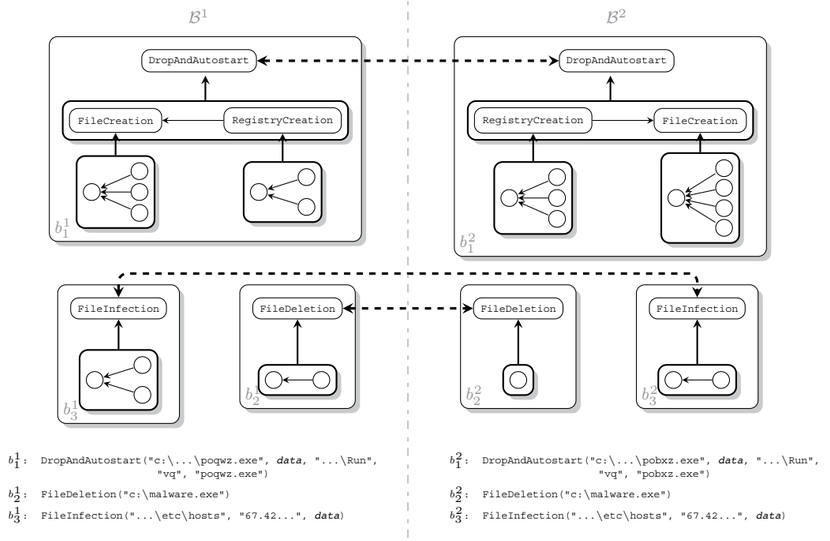



Figure 4: High-level behavior clustering.

$\mathcal{C}_1 = \{b^1_1, b^2_1\}$ represents `DropAndAutostart` behaviors, $\mathcal{C}_2 = \{b^1_2, b^2_2\}$ represents `FileDeletion` behaviors, and $\mathcal{C}_3 = \{b^1_3, b^2_3\}$ represents `FileInfection` behaviors.

**Behavior Comparison:** Our clustering algorithm requires a sub-algorithm, *isomorphic*, to compare two behaviors. Intuitively, we perform this comparison by *normalizing* the graphs corresponding to each behavior and then checking whether the resulting normalized graphs are isomorphic. There is an important advantage in comparing the behavior graphs rather than their high-level summaries: nondeterminism in a malicious program typically affects the summary of the behavior, but not the low-level operations used to achieve the behavior. Therefore, this approach is more resilient to nondeterminism and performs a more thorough comparison, eventually yielding more precise results.

The normalization we perform on each graph mainly consists of abstracting away details of the behavior that are likely affected by nondeterminism. System call arguments that represent resource names are replaced by constants that denote their type. For example, we use a different constant for each file and registry type. Sequences of system calls that operate sequentially on the same resource are replaced with a single, *batch* call that is semantically identical. Finally, we ignore system calls whose effects are later *killed*, i.e. overwritten or otherwise reversed. In this way, our normalization step produces more succinct graph representations of the malware's behavior that are largely independent of common forms of nondeterminism.

After normalizing two graphs for comparison, we use the VFlib2 graph isomorphism algorithm [6]. Although isomorphism is a difficult problem and may be inefficient to compute on large graphs, we point out that the normalized behavior graphs resulting from real-world programs are typically quite small, comprising no more than a few dozen nodes.

#### 4.2.3 Behavior Generalization

After clustering, we have several sets of behaviors grouped by semantic similarity but still differing in certain details. For example, when we build clusters we group together behaviors that differ in the specific resources they identify. The goal of behavior generalization is to produce a single canonical behavior that represents all of the members of a given cluster, as well as variations of the members that are likely to result from other executions of the malware. In terms of the definitions presented in Section 4.1, behavior generalization produces high-level behaviors with arguments constructed to accurately represent the resources modified by observed executions, while generalizing to potential future executions.

Algorithm 2 presents *Generalize*, our procedure for generalizing a behavior cluster. Intuitively, generaliza-

**Algorithm 1** *Cluster*$(\mathcal{B}, \mathcal{B}^{\max})$

**Require:** $\mathcal{B}$ is a set of high-level behavior traces $\{\mathcal{B}^1, \mathcal{B}^2, \ldots, \mathcal{B}^m\}$
$\mathcal{B}^{\max}$ is the high-level behavior trace containing the maximum number of high-level behaviors
**Result:** A set of clusters of high-level behaviors of $\{\mathcal{B}^1, \mathcal{B}^2, \ldots, \mathcal{B}^m\}$
$\mathcal{C} \leftarrow \varnothing$
**for** $b_j^{\max} \in \mathcal{B}^{\max}$ **do**
  add new cluster $\{b_j^i\}$ to $\mathcal{C}$
**end for**
**for all** $\mathcal{B}^i \in \mathcal{B}/\mathcal{B}^{\max}$ **do**
  {Traces are enumerated in the order of collection.}
  **for all** $b_j^i \in \mathcal{B}^i$ **do**
    {Behaviors are enumerated in execution order.}
    **for all** $\mathcal{C}_k \in \mathcal{C}$ **do**
      **if** $isomorphic(b_j^i, b_k)$ where $b_k$ is a behavior in $\mathcal{C}_k$ **then**
        $\mathcal{C}_k \leftarrow \mathcal{C}_k \cup \{b_j^i\}$
      **end if**
    **end for**
    **if** $b_j^i$ is not in any cluster **then**
      add new cluster $\{b_j^i\}$ to $\mathcal{C}$
    **end if**
  **end for**
**end for**
**return** $(\mathcal{C})$

**Algorithm 2** *Generalize*$(C, \mathcal{G}, \delta)$

**Require:** $C$ is a cluster of behaviors that differ only in argument values, $\mathcal{G}$ is a set of generalization rules, $\delta$ is the density threshold.
**Result:** A generalized high-level behavior.
{Loop through all arguments for behaviors in cluster $C$}
**for** $i = 0$ **to** $|args(C_0)|$ **do**
  $A_i \leftarrow \varnothing$
  {Gather all values for current argument}
  **for** $c$ **in** $C$ **do**
    $A_i \leftarrow A_i \cup args_i(c)$
  **end for**
  {Generate PFSA that captures argument values}
  $(V, E) \leftarrow \text{PFSA}(A_i)$
  {Find dense regions in the PFSA}
  **for** $(n_1, n_2)$ **in** $V \times V - \{(n, n) \mid n \in V\}$ **do**
    **if** $\neg idom(e_1, e_2)$ **or** $\neg ipdom(n_2, n_1)$ **or** $numpaths(n_1, n_2) < \delta$ **then**
      **continue**
    **end if**
    **for** $r$ **in** $\mathcal{G}$ **do**
      $E' \leftarrow r(paths(n_1, n_2))$
      $E \leftarrow (E - paths(n_1, n_2)) \cup E'$
    **end for**
  **end for**
  {Build regular expression for the current arguments}
  $G_i \leftarrow regexp(E)$
**end for**
{Return new behavior with type matching $C$, and generalized reg. exp. arguments}
**return** $name(C_0)(G_0, \ldots, G), 0 \leq n < |args(C_0)|$

tion is performed on each high-level behavior argument individually, and the individual results are eventually combined to produce the generalized behavior. Because each cluster member represents the same high-level behavior, and therefore has the same number of arguments as the others, we are assured that all of the relevant information is included in the generalization. Furthermore, because all arguments for the behaviors that we are interested in have straightforward canonical representations as strings, the problem of generalizing each argument can be reduced to the problem of generalizing sets of strings. *Generalize* proceeds in this vein, iterating over each argument for the behaviors in a given cluster $C$. After collecting each string for a given argument in a set $A_i$, a probabilistic finite-state automaton (PSFA) that accepts all of the strings in $A_i$ is constructed using the *simulated beam annealing* algorithm [17]. By merging states that are probabilistically very similar, the resulting automaton accepts a superset of $A_i$, thus resulting an initial generalization.

After building the PFSA, certain regions of the state transition diagram are examined for reduction using a set $\mathcal{G}$ of *generalization rules*, which are templates for generating regular expressions that overapproximate high-level behavior arguments. We refer to a *single-entry single-exit region* as one whose entry is composed of a node $n_1$ that is the immediate dominator of the exit node $n_2$, which is the immediate postdominator of $n_1$. Furthermore, we require that the number of paths between $n_1$ and $n_2$ be at least $\delta$. The actual value of $\delta$ is estimated empirically. This information is represented in Algorithm 2 with the relations $idom_E$ and $ipdom_E$, as well as the function $numpaths_E$. When a suitable single-entry single-exit region is found, each rule in $\mathcal{G}$ is applied in an attempt to generalize it. The generalization rules that we use have been chosen on the basis of experience and consider information such as the number of paths in the region, the probabilities associated with the paths, the lengths of the paths, and the characters composing the strings associated with each path. If a rule is able to generalize the region, then it returns a smaller set of edges that are used to replace the original region. Otherwise, the rule returns the original region, and the next rule is applied. After all rules in $\mathcal{G}$ have been applied, a regular expression is built from the resulting PFSA, which is eventually used as an argument in the final generalized behavior. The final behavior is represented in Algorithm 2 by $name(C_0)(G_0, \ldots, G_n)$. Here, $name(C_0)$ returns the behavior name of the high-level behavior $C_0$, which is used to build the final generalized behavior from



```
1 DropAndAutostart("c:\windows\...poagp.exe",data,"...Windows\CurrentVersion\Run","vq","poagp.exe")
2 DropAndAutostart("c:\windows\...pobxz.exe",data,"...Windows\CurrentVersion\Run","vq","pobxz.exe")
3 DropAndAutostart("c:\windows\...pocra.exe",data,"...Windows\CurrentVersion\Run","qv","pocra.exe")
4 DropAndAutostart("c:\windows\...pomfq.exe",data,"...Windows\CurrentVersion\Run","vq","pomfq.exe")
5 DropAndAutostart("c:\windows\...pommp.exe",data,"...Windows\CurrentVersion\Run","qv","pommp.exe")
6 DropAndAutostart("c:\windows\...popwz.exe",data,"...Windows\CurrentVersion\Run","qv","popwz.exe")
7 DropAndAutostart("c:\windows\...pouwk.exe",data,"...Windows\CurrentVersion\Run","vq","pouwk.exe")
```

Figure 5: Sample cluster grouping seven different occurrences of the `DropAndAutostart` behavior manifested by our sample malware (the corresponding graphs are omitted for conciseness).

the individual argument generalizations.

As an illustration of this algorithm, consider the cluster presented in Figure 5. We apply the PFSA algorithm to the first argument to arrive at the minimal automaton shown in Figure 6. The automaton contains a single-entry single-exit region with several paths, as highlighted in the figure, that encodes the variable substring of the filename. One of the generalization rules that we use is triggered by the fact that this region is *dense*, i.e. it contains many paths from entry to exit, as well as the fact that it contains only alphabetic characters. Thus, it returns a single edge labeled [[: `alpha` :]]{3}, which is a wildcard sequence that denotes all alphabetic strings of length three. The generalized PFSA results in the regular expression `c:\windows\po`[[: `alpha` :]]{3}`.exe`, which is capable of identifying all the names of the files that our sample malicious program could touch on the system. After applying *Generalize* to all arguments of `DropAndAutostart`, we obtain a generic model of the cluster behavior represented by `DropAndAutostart`("`c:\windows\po`[[: `alpha` :]]{3}`.exe`", *data*, "...`Windows\CurrentVersion\Run`", "(vq|qv)").

#### 4.2.4 Generating Concrete Remediation Procedures

Each generalized high-level behavior must be remediated differently. Our approach to generating executable remediation procedures may be understood conceptually in two parts. First, the generalized high-level behaviors for each cluster are used to construct an approximate infection relation $R$ as discussed in Section 4.1. Then, we use a generic procedure that scans the infection relation, and changes the state of the system based on the contents of each entry. When constructing the infection relation, our procedure uses a model of a clean, bare installation of the operating system installed on the machine for the first system state component of each tuple. The use of a bare installation enables us to remediate infected system resources up to the correct service pack installed on the system, but not personal or application-specific resources.

The remainder of this section details the way that specific high-level behaviors are translated into entries in the abstract infection relation, as well as the way that the abstract infection relation is used to generate a concrete (executable) remediation procedure.

**Algorithm 3** *Remediate*$(S, R)$

$(S_{abs}, N_{rem}, S_{add}, S_{mut}) \leftarrow (S_{abs}, N_{rem}, S_{add}, S_{mut}) \in R$ such that $S$ has the same operating system version as $S_{abs}$
**for** $s$ **in** $S_{add}$ **do**
  **cases** $s$:
    $(name, data)$ : **if** file $name$ exists, with contents matching $data$ **then** remove $name$.
    $((key, value), data)$ : **if** $(key, value)$ exists with contents matching $data$ **then** remove $(key, value)$.
    $((file, key, value), (data, regdata))$ : **if** $file$ exists and is a suffix of some element of $D_{regdata}$ that also exists in a key matching $(key, value)$ **then** remove $file$ and $(key, value)$.
    $((file, procname), data)$ : **if** $procname$ and $(file, data)$ exist matching $file$, $data$, $procname$ **and** $procname$ is a suffix of $file$ **then** remove $file$ and kill $procname$.
  **end cases**
**end for**
**for** $i$ **in** $I_{mut}$ **do**
  **cases** $i$:
    $(file, data)$ : Remove $(file, data)$ and replace it with $(file, data') \in \beta(S)$
    $((key, value), data)$ : Remove $((key, value), data)$ and replace it with $((key, value), data) \in \beta(S)$.
  **end cases**
**end for**

**Newly-Created Resources:** Remediating resources that are created by malware is straightforward, because the remediation procedure only needs information regarding the names and data of newly-created resources to completely remove the corresponding resources from the system. Our remediation procedures are capable of removing files and registry keys. To account for the possibility that the infection could create resources that were not observed in a high-level behavior trace during analysis, we instead use generalized high-level behaviors in the infection relation $R$.

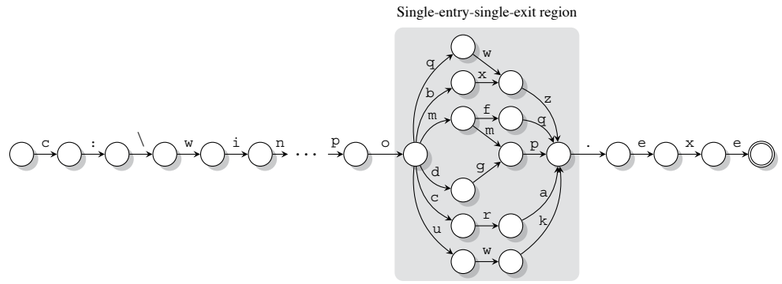



Figure 6: A fragment of the minimized automaton constructed to generalize the first argument of the `DropAndAutostart` behavior, starting from the occurrences of the argument reported in Figure 5.

For the high-level file creation behavior `FileCreation`$(name, data)$, we find the resource for *name* and *data* and append this pair to $S_{add}$. Similarly, for the high-level registry creation behavior `RegistryCreation` $(key, value, data)$, we associate the key/value pair to the corresponding data and add them as a pair to $S_{add}$. As shown in Algorithm 3, the remediation procedure processes these entries in the infection relation $R$ by checking for the existence of the resource names on the system and removing them if they exist with the contents specified by $R_\alpha$.

Remediating the `DropAndAutostart` and `DropAndExecute` behaviors is more complicated, as doing so involves multiple resources that are related in a constrained manner. To handle a high-level behavior of the form:

`DropAndAutostart`$(file, data, key, value, regdata)$

we group the resource names: *file, key, value* together as a compound resource name for a new element in $S_{add}$, and group *data* and *regdata* together for the corresponding data component. The remediation procedure acts on such an entry by scanning system resources for names that match the file name and registry key/value pairs. If a match is found, the corresponding resources are removed only if the concrete filename is a suffix of the concrete registry data and the concrete data matches the abstract data.

For example, when the procedure encounters the generalized `DropAndAutostart` from Figure 5, it will augment $S_{add}$ with the following resource:

(`c : \windows\po[[: alpha :]]{3}\.exe`,
(...\`CurrentVersion`, `Run`),
($data$, `po[[: alpha :]]{3}\.exe`))

The remediation procedure will then search the system for a file that matches `c : \windows\po[[: alpha :]]{3}.exe`, as well as the registry key (...\`CurrentVersion`, `Run`), and will remove the resources only if the value of the registry key matches the name of any file that matches the regular expression.

**Infected Resources:** Remediating infected resources is more challenging than newly-created resources. In general, it is not possible to know the contents of a file before infection takes place, so it is not possible to restore their contents to a clean state. The exception to this fact is with operating system files, which are common to all systems and can thus be known to the remediation procedure *a priori*.

A naive approach to remediating high-level `FileInfection`$(name, region, data)$ behaviors would be to replace the entire file with the corresponding file in the bare operating system. However, uninfected regions of data may be removed by this technique, which could result in the loss of important system data, or leave the system in an inconsistent state. To avoid this circumstance, high-level behavior traces keep track of uninfected regions *regions* in addition to file name *file* and infected data *data*. We update the $S_{mut}$ component of $R$ to account for a `FileInfection` behavior only if there is an actual file in the clean operating system state whose name matches the *file*. In this case, $S_{mut}$ is updated with the contents of *file* in the bare operating system state, modified by preserving the portions listed in *regions* and overwriting the rest with *data*. As indicated in Algorithm 3, when the remediation procedure finds *file*, it replaces the infected regions with a pristine copy from the bare operating system.

Similarly, when a high-level `RegistryInfection`$((key, value), data)$ behavior is encountered, and it is determined that a counterpart of $(key, value)$ exists in the bare operating system, $S_{mut}$ is modified by adding the key/value pair together with the modified data to the list of infected resources. As with infected files, Algorithm 3 remediates these resources by locating a pristine copy of $(key, value)$ in the bare operating system and replacing the infected resource with it.

**Deleted Resources:** Currently, most malware is written with the intent of leveraging infected systems to perpetrate profitable, albeit illicit, activities. Therefore, it is very rare to see malware removing system resources, as doing so would render the system useless for money-making activities. For this reason, our remediation procedures do not handle deleted resources.

# 5 Evaluation

We applied our remediation procedure generation algorithm to over two hundred malware samples collected in the wild. We evaluated the quality of the generated procedures with respect to two metrics: false positives and false negatives. A false positive occurs when a resource is mistakenly identified as being part of a malware infection and subsequently remediated. A false negative occurs when a resource that was actually involved in an infection is not identified and left untouched by the remediation procedure. The results of our evaluation testify to the effectiveness of our technique: we observed a low false negative rate, with more than 98% of the malicious resources successfully remediated, and only one false positive was encountered. Finally, we compare our results to the remediation capabilities of the three commercial products that performed best in previous experiments [15].

## 5.1 Experimental Setup

Our experiments were performed over a corpus of 200 malicious programs, obtained through our own honeypot, and a web crawler that crawls known malicious domains for executable files. Several traces for each sample were collected by executing it in multiple distinct environments. To extract a wide range of behaviors from each sample, we modified the environments along a variety of dimensions, including locale, timezone, and the set of installed applications. Specifically, for each sample we performed the following steps:

1. Execute the sample three times in five different environments, collecting a system call trace for each execution. Apply the algorithm described in Section 4.2 to generate a remediation procedure from the collected data.
2. Infect twenty-five test environments, all of them distinct from those used to collect traces, with the sample.
3. Execute the generated remediation procedure in each test environment.
4. Compare the remediated state to the original (clean) state. Tally the false positives and false negatives.

Although we do not attempt to extract all possible execution paths from the malware, this strategy allows us to observe a reasonable range of malware behavior in various settings.

## 5.2 False Negatives

Figure 7 compares the false negative rate of our automatically-generated remediation procedures with the three top-rated commercial malware detectors evaluated in [15]: Nod32 Anti-Virus 3.0, Panda Anti-Virus 9.0.5, and Kaspersky Anti-Virus 2009. The graph depicts the average number of malicious resources that were remediated over the entire malware corpus. Resources are divided into three categories: files, registry keys, and processes. Each of these classes is further divided into two subcategories: primary and ancillary. Primary resources are composed of executable files, registry keys that activate process creation, and processes that arise from files dropped or infected by the malware sample. Roughly, we argue that all other resources are not as critical to the security of the system, and are thus considered ancillary.

For the majority of these categories and subcategories, our remediation procedures are more complete than commercial anti-malware products. For example, our procedures were able to remediate more than 99% of the primary file resources, whereas the best commercial product we tested reached only 82% in this subcategory. Similarly, our procedures remediated 99% of primary registry activities, while commercial products did not exceed 86%. Furthermore, while ancillary objects are often ignored by commercial remediation procedures, our procedures remediated 95% of ancilliary files and 98% of ancilliary registry activities. The portion of file and registry resources that were not remediated by our procedures correspond to behaviors that were never observed while collecting traces. This illustrates the primary limitation of our dynamic analysis-based approach and highlights a clear avenue for improvement in future work. Finally, our procedures remediated 100% of primary process resources. However, the performance on ancillary processes is significantly lower. This is a result of the fact that our processes do not have access to enough information to discern a benign process from a process spawned by the malware using a pre-existing benign file.

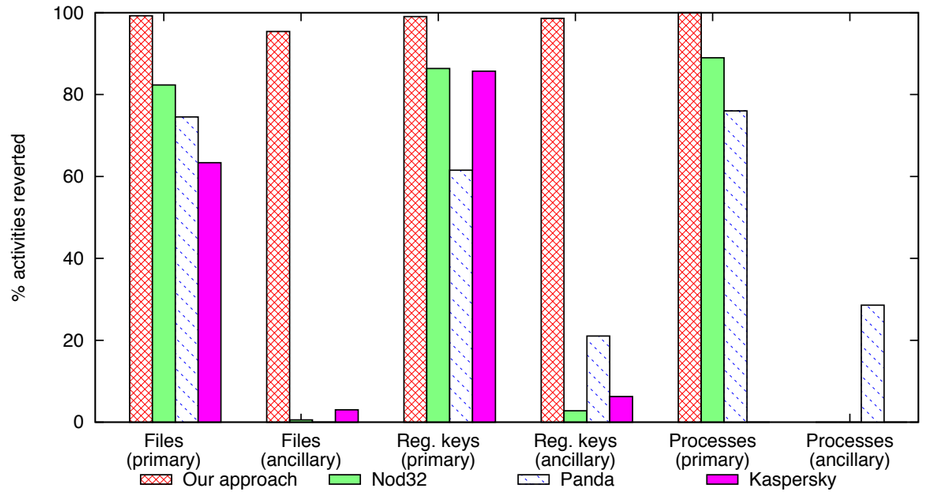



Figure 7: Comparison of the completeness of our automatically generated remediation procedures with the completeness of the procedures employed in three top-rated commercial malware detectors.

### 5.3 False Positives

To quantify false positives, we compared the set of resources affected by each malware sample in each test environment with the set of resources our procedures remediated in each test environment. Any remediated resource not affected by the corresponding malware sample in at least one trace is considered a false positive. We found that only one of our procedures produced any false positives. The cause of this false positive, not surprisingly, was a high-level behavior argument specified by a very general regular expression. This implies that the nondeterminism demonstrated by the corresponding malware sample was too complex to be easily described by a regular language. Thus, one area for future work is utilizing more expressive language classes, such as context-free grammars, for generalizing argument values.

## 6 Discussion

We are aware of some limitations of our system. Some of these limitations could be exploited by attackers to cause the system to produce remediation procedures that are of limited value. In this section, we discuss these limitations and present some solutions that we will investigate in the future to address the limitations.

We constructed the models that we use to detect high-level behaviors by leveraging years of experience in malware analysis, and we carefully tested all models to ensure that they cannot be evaded. However, since we cannot prove that these models are perfect, we must take into account the possibility that attackers could find new ways to perform some high-level malicious activities without being detected. Moreover, in our proof-of-concept implementation, multiple execution traces are obtained by executing the same malware in several different operating system configurations. If attackers introduced dangerous behaviors to their malicious programs that are not triggered in our monitoring environment, then the resulting procedure would not be able to remediate such behaviors. Clearly, one area for future work is in expanding the coverage of the dynamic behavioral analysis. While our approach covers some of the potential behavior of the sample, more sophisticated techniques [12, 21] can be applied to increase the likelihood that all relevant paths through the malware are explored.

The high-level behaviors observed in multiple execution traces are clustered to identify the instances of the same behavior. If the clusters we generate did not include all the instances of the same behavior, or if they included instances of different behaviors, then the remediation procedures constructed by generalizing the behaviors associated to each cluster would be too specific or too generic. An attacker could write malicious programs that manifest certain behaviors to break the clustering. Similarly, the regular expressions used by our remediation procedures to identify affected resources are generalized heuristically. Attackers could develop malicious programs that affect resources in a way that induces us to perform very aggressive generalization (e.g. create files with random names anywhere in the file system) and thus to generate remediation procedures that remove benign files. We plan to address these problems in the future. One approach is to introduce a feedback loop while clustering behaviors and generating regular expressions to validate the quality of the results. This feedback loop would repeat the process until no further progress can be made. Finally, we assert that it is not possible to cause our algorithm to generate a procedure that modifies existing files in a harmful way. This follows from the fact that system files are only ever restored to their original state by the procedure, not modified.

We currently generate a remediation procedure for each malware sample we analyze. We plan to extend our system to generate remediation procedures that cover more than one malware sample. For example, it would be useful to generate remediation procedures that are capable of operating on all samples for a given malware family. Because the generated procedures will likely have to account for a much higher degree of nondeterminism than those that target only a single sample, additional care must be taken to ensure that the high-level behaviors models are not too general, thus resulting in false positives.

## 7 Conclusion

In this paper, we have presented a technique for automatically generating malware remediation procedures. Given a malware binary, our system produces executable code that removes the harmful effects of executing that malware on a system. We use dynamic analysis and *behavior generalization* to account for the difficulties posed by real malware, thus allowing our procedures to effectively remediate many possible executions of the malware without witnessing the actual infection take place. This contribution represents a major break with previous automatic remediation techniques, which required detailed information about the particular infection being targeted. We implemented our technique and evaluated its effectiveness on more than 200 malware binaries. The performance of our prototype is quite good: on average, 98% of the harmful effects are remediated, and we encountered only a single false positive. In the future, we plan to build on this work by extending it to work on entire families, as well as exploring more precise techniques for generalizing observed malware behaviors.

## References

[1] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *15th European Institute for Computer Antivirus Research (EICAR) Annual Conference*, Hamburg, Germany, Apr. 2006.

[2] U. Bayer, P. Milani, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. `http://fabrice.bellard.free.fr/qemu/`.

[4] M. Christodorescu, C. Kruegel, and S. Jha. Mining specifications of malicious behavior. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Dubrovnik, Croatia, 2007.

[5] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[6] P. Foggia. The vflib graph matching library, version 2.0. `http://amalfi.dis.unina.it/graph/db/vflib-2.0/`.

[7] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the future: A framework for automatic malware removal and system repair. In *22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[8] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, 2009.

[9] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *IEEE Symposium on Security and Privacy*, Oakland, California, 2006.

[10] Z. Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *19th Annual Computer Security Applications Conference (ACSAC)*, 2003.

[11] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2008.

[12] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, Oakland, California, 2007.

[13] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[14] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, Oakland, California, 2005.

[15] E. Passerini, R. Paleari, and L. Martignoni. How good are malware detectors at remediating infected systems? In *6th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Como, Italy, July 2009.

[16] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems*, 30(5):25.1–25.54, Aug. 2008.

[17] A. Raman, P. Andreae, and J. Patrick. A beam search algorithm for PFSA inference. *Pattern Analysis and Applications*, 1(2):121–129, 1998.

[18] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2008.

[19] W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *12th Symposium on Network and Distributed Systems Security (NDSS)*, 2005.

[20] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *14th USENIX Security Symposium*, Baltimore, MD, 2005.

[21] H. Yin, D. Song, M. Egele, E. Kirda, and C. Kruegel. Panorama: Capturing system-wide information flow for malware detection and analysis. In *14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, 2007.

# Re: CAPTCHAs – Understanding CAPTCHA-Solving Services in an Economic Context

*Marti Motoyama, Kirill Levchenko, Chris Kanich, Damon McCoy,*
*Geoffrey M. Voelker and Stefan Savage*
*University of California, San Diego*
{*mmotoyam, klevchen, ckanich, dlmccoy, voelker, savage*}*@cs.ucsd.edu*

## Abstract

Reverse Turing tests, or CAPTCHAs, have become an ubiquitous defense used to protect open Web resources from being exploited at scale. An effective CAPTCHA resists existing mechanistic software solving, yet can be solved with high probability by a human being. In response, a robust solving ecosystem has emerged, reselling both automated solving technology and real-time human labor to bypass these protections. Thus, CAPTCHAs can increasingly be understood and evaluated in purely economic terms; the market price of a solution *vs* the monetizable value of the asset being protected. We examine the market-side of this question in depth, analyzing the behavior and dynamics of CAPTCHA-solving service providers, their price performance, and the underlying labor markets driving this economy.

## 1 Introduction

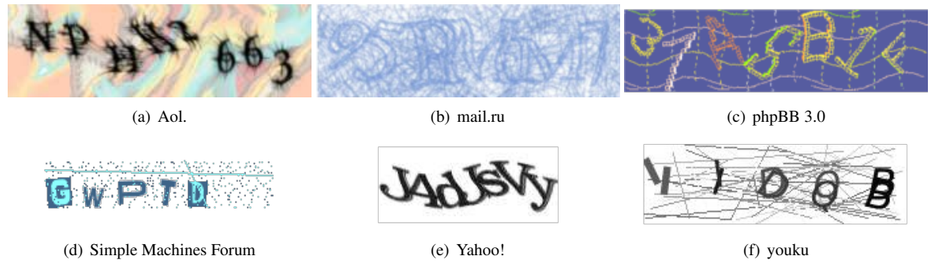
Questions of Internet security frequently reflect underlying economic forces that create both opportunities and incentives for exploitation. For example, much of today's Internet economy revolves around advertising revenue, and consequently, a vast array of services—including e-mail, social networking, blogging—are now available to new users on a basis that is both free and largely anonymous. The implicit compact underlying this model is that the users of these services are *individuals* and thus are effectively "paying" for services indirectly through their unique exposure to ad content. Unsurprisingly, attackers have sought to exploit this same freedom and acquire large numbers of resources under singular control, which can in turn be monetized (e.g., via thousands of free Web mail accounts for sourcing spam e-mail messages).

CAPTCHAs were developed as a means to limit the ability of attackers to scale their activities using automated means. In its most common implementation, a CAPTCHA consists of a visual challenge in the form of alphanumeric characters that are distorted in such a way that available computer vision algorithms have difficulty segmenting and recognizing the text. At the same time, humans, with some effort, have the ability to decipher the text and thus respond to the challenge correctly. Today, CAPTCHAs of various kinds are ubiquitously deployed for guarding account registration, comment posting, and so on.

This innovation has, in turn, attached value to the problem of solving CAPTCHAs and created an industrial market. Such commercial CAPTCHA solving comes in two varieties: automated solving and human labor. The first approach defines a technical arms race between those developing solving algorithms and those who develop ever more obfuscated CAPTCHA challenges in response. However, unlike similar arms races that revolve around spam or malware, we will argue that the underlying cost structure favors the defender, and consequently, the conscientious defender has largely won the war.

The second approach has been transformative, since the use of human labor to solve CAPTCHAs effectively side-steps their design point. Moreover, the combination of cheap Internet access and the commodity nature of today's CAPTCHAs has globalized the solving market; in fact, wholesale cost has dropped rapidly as providers have recruited workers from the lowest cost labor markets. Today, there are many service providers that can solve large numbers of CAPTCHAs via on-demand services with *retail* prices as low as $1 per thousand.

In either case, we argue that the security of CAPTCHAs can now be considered in an economic light. This property pits the underlying cost of CAPTCHA solving, either in amortized development time for software solvers or piece-meal in the global labor market, against the value of the asset it protects. While the very existence of CAPTCHA-solving services tells us that the value of the associated assets (e.g., an e-mail account) is worth more to some attackers than the cost of solving the CAPTCHA, the overall shape of the market is poorly understood. Ab-



(a) Aol. (b) mail.ru (c) phpBB 3.0

(d) Simple Machines Forum (e) Yahoo! (f) youku

Figure 1: Examples of CAPTCHAs from various Internet properties.

sent this understanding, it is difficult to reason about the security value that CAPTCHAs offer us.

This paper investigates this issue in depth and, where possible, on a empirical basis. We document the commercial evolution of automated solving tools (particularly via the successful Xrumer forum spamming package) and how they have been largely eclipsed by the emergence of the human-based CAPTCHA-solving market. To characterize this latter development, our approach is to engage the retail CAPTCHA-solving market on both the supply side and the demand side, as both a client and as "workers for hire." In addition to these empirical measurements, we also interviewed the owner and operator of a successful CAPTCHA-solving service (MR. E), who has provided us both validation and insight into the less visible aspects of the underlying business processes.[1] In the course of our analysis, we attempt to address key questions such as which CAPTCHAs are most heavily targeted, the rough solving capacity of the market leaders, the relationship of service quality to price, the impact of market transparency and arbitrage, the demographics of the underlying workforce and the adaptability of service offerings to changes in CAPTCHA content. We believe our findings, or at least our methodology, provide a context for reasoning about the net value provided by CAPTCHAs under existing threats and offer some directions for future development.

The remainder of this paper is organized as follows: Section 2 reviews CAPTCHA design and provides a qualitative history and overview of the CAPTCHA-solving ecosystem. Next, in Section 3 we empirically characterize two automated solver systems, the popular Xrumer package and a specialized reCaptcha solver. In Sections 4 and 5 we then characterize today's human-powered CAPTCHA-solving services, first describing our data collection approach and then presenting our experiments to measure key qualities such as response time, accuracy, and capacity. Section 6 describes the demographics of the CAPTCHA-solving labor pool. Finally, we discuss the implications of our results in Section 7 along with potential directions for future research.

## 2 Background

The term "CAPTCHA" was first introduced in 2000 by von Ahn *et al.* [21], describing a test that can differentiate humans from computers. Under common definitions [4], the test must be:

- Easily solved by humans,
- Easily generated and evaluated, but
- *Not* easily solved by computer.

Over the past decade, a number of different techniques for generating CAPTCHAs have been developed, each satisfying the properties described above to varying degrees. The most commonly found CAPTCHAs are visual challenges that require the user to identify alphanumeric characters present in an image obfuscated by some combination of noise and distortion.[2] Figure 1 shows examples of such visual CAPTCHAs. The basic challenge in designing these obfuscations is to make them easy enough that users are not dissuaded from attempting a solution, yet still too difficult to solve using available computer vision algorithms.

The issue of usability has been studied on a functional level—focusing on differences in expected accuracy and response time [3, 19, 22, 26]—but the ultimate effect of CAPTCHA difficulty on legitimate goal-oriented users is not well documented in the literature. That said, Elson *et al.* provide anecdotal evidence that "even relatively simple challenges can drive away a substantial number of potential customers" [9], suggesting CAPTCHA design reflects a real trade-off between protection and usability.

The second challenge, defeating automation, has received far more attention and has kicked off a competition of sorts between those building ever more sophisticated algorithms for breaking CAPTCHAs and those creating new, more obfuscated CAPTCHAs in response [7, 11, 16, 17, 18, 25]. In the next section we examine this issue in more depth and explain why, for economic reasons, automated solving has been relegated to a niche status in the open market.

Finally, an alternative regime for solving CAPTCHAs is to outsource the problem to human workers. Indeed, this labor-based approach has been commoditized and today a broad range of providers operate to buy and sell CAPTCHA-solving service in bulk. We are by no means the first to identify the growth of this activity. In particular, Danchev provides an excellent overview of several CAPTCHA-solving services in his 2008 blog post "Inside India's CAPTCHA solving economy" [5]. We are, however, unaware of significant quantitative analysis of the solving ecosystem and its underlying economics. The closest work to our own is the complementary study of Bursztein *et al.* [3] which also uses active CAPTCHA-solving experiments, but is focused primarily on the issue of CAPTCHA difficulty rather than the underlying business models.

## 3 Automated Software Solvers

From the standpoint of an adversary, automated solving offers a number of clear advantages, including both near-zero marginal cost and near-infinite capacity. At a high level, automated CAPTCHA solving combines *segmentation* algorithms, designed to extract individual symbols from a distorted image, with basic optical character recognition (OCR) to identify the text present in CAPTCHAs. However, building such algorithms is complex (by definition, since CAPTCHAs are designed to evade existing vision techniques), and automated CAPTCHA solving often fails to replicate human accuracy. These constraints have in turn influenced the evolution of automated CAPTCHA solving as it transitioned from a mere academic contest to an issue of commercial viability.

### 3.1 Empirical Case Studies

We explore these issues empirically through two representative examples: Xrumer, a mature forum spamming tool with integrated support for solving a range of CAPTCHAs and reCaptchaOCR, a modern specialized solver that targets the popular reCaptcha service.

#### Xrumer

Xrumer [24] is a well-known forum spamming tool, widely described on "blackhat" SEO forums as being one of the most advanced tools for bypassing many different anti-spam mechanisms, including CAPTCHAs. It has been commercially available since 2006 and currently retails for $540, and we purchased a copy from the author at this price for experimentation. While we would have liked to include several other well known spamming tools (SEnuke, AutoPligg, ScrapeBox, etc), the cost of these packages range from $97 to $297, which would render this study prohibitively expensive.

Xrumer's market success in turn led to a surge of spam postings causing most service providers targeted by Xrumer to update their CAPTCHAs. This development kicked off an "arms race" period in Xrumer's evolution as the author updated solvers to overcome these obstacles. Version 5.0 of Xrumer was released in October of 2008 with significantly improved support for CAPTCHA solving. We empirically verified that 5.0 was capable of solving the default CAPTCHAs for then current versions of a number of major message boards, including: Invision Power Board (IPB) version 2.3.0, phpBB version 3.0.2, Simple Machine Forums (SMF) version 1.1.6, and vBulletin version 3.6. These systems responded in kind, and when we installed versions of these packages released shortly after Xrumer 5.0 (in particular, phpBB and vBulletin) we verified that their CAPTCHAs had been modified to defeat Xrumer's contemporaneous solver. Today, we have found that the only major message forum software whose default CAPTCHA Xrumer can solve is Simple Machines Forum (SMF).

With version 5.0.9 (released August 2009), Xrumer added integration for human-based CAPTCHA-solving services: Anti-Captcha (an alias for Antigate) and CaptchaBot. We take this as an indication that the author of Xrumer found the ongoing investment in CAPTCHA-solving software to be insufficient to support customer requirements.[3] That said, Xrumer can be configured to use a hybrid software/human based approach where Xrumer detects instances of CAPTCHAs vulnerable to its automated solvers and uses human-based solvers otherwise. In the current version of Xrumer (5.0.12), the CAPTCHA-related development seems to focus on supporting automatic navigation and CAPTCHA "extraction" (detecting the CAPTCHA and identifying the image file to send to the human-based CAPTCHA-solving service) of more Web sites, as well as evading other anti-spam techniques.

[1] By agreement, we do not identify MR. E or the particular service he runs. While we cannot validate all of his statements, when we tested his service empirically our results for measures such as response time, accuracy, capacity and labor makeup were consistent with his reports, supporting his veracity.

[2] There exists a range of non-textual and even non-visual CAPTCHAs that have been created but, excepting Microsoft's *Asirra* [9], we do not consider them here as they play a small role in the current CAPTCHA-solving ecosystem.

[3] The developers of Xrumer have recently been advertising enhanced CAPTCHA-solving functionality in their forthcoming "7.0 Elite" version (including support for reCaptcha), but the release date has been steadily postponed and, as of this writing (June 2010), version 5.0.12 is the latest.

When compared with developers targeting "high-value" CAPTCHAs (e.g., reCaptcha, Microsoft, Yahoo, Google, etc.), Xrumer has mostly targeted "weaker" CAPTCHAs and seems to have a policy of only including highly efficient and accurate software-based solvers. In our tests, all but one included solver required a second or less per CAPTCHA (on a netbook class computer with only a 1.6-GHz Intel Atom CPU) and had an accuracy of 100%. The one more difficult case was the solver for the phpBB version 3 forum software with the GD CAPTCHA generator and foreground noise. In this case, Xrumer had an accuracy of only 35% and required 6–7 seconds per CAPTCHA to execute.

**reCaptchaOCR**

At the other end of the spectrum, we obtained a specialized solver focused singularly on the popular reCaptcha service. Wilkins developed the solver as a proof of concept [23]. The existence of this OCR-based reCaptcha solver was reported in a blog posting on December 15, 2009 [6]. Although developed to defeat an earlier version of reCaptcha CAPTCHAs (Figure 2a), reCaptchaOCR was also able to defeat the CAPTCHA variant in use at the time of release (Figure 2b). Subsequently, reCaptcha changed their CAPTCHA-generation code again to the version as of this writing (Figure 2c). The tool has not been updated to solve this new variant.

We tested reCaptchaOCR on 100 randomly selected CAPTCHAs of the early 2008 variant and 100 randomly selected CAPTCHAs of the late 2009 variant. We scored the answers returned using the same algorithm that reCaptcha uses by default. reCaptcha images consist of two words, a control word for which the correct solution is known, and the other a word for which the solution is unknown (the service is used to opportunistically implement human-based OCR functionality for difficult words). By default reCaptcha will mark a solution as correct if it is within an edit distance of one of the control word. However, while we know the ground truth for both words in our tests, we do not know which was the control word. Thus, we credited the solver with half a correct solution for each *word* it solved correctly in the CAPTCHA, reasoning that there was a 50% chance of each word being the control word.

We observed an accuracy of 30% for the 2008-era test set and 18% for the 2009-era test set using the default setting of 613 iterations,[4] far lower than the average human accuracy for the same challenges (75–90% in our experiments).

Finally, we measured the overhead of reCaptchaOCR. On a laptop using a 2.13-GHz Intel Core 2 Duo each solution required an average of 105 seconds. By reducing the number of iterations to 75 we could reduce the solving time to 12 seconds per CAPTCHA, which is in line with the response time for a human solver. At this number of iterations, reCaptchaOCR still achieved similar accuracies: 29% for the 2008-era CAPTCHAs and 17% for the 2009-era CAPTCHAs.

[4]The solver performs multiple iterations and uses the majority solution to improve its accuracy.

## 3.2 Economics

Both of these examples illustrate the inherent challenges in fielding commercial CAPTCHA-solving software.

While the CAPTCHA problem is often portrayed in academia as a technical competition between CAPTCHA designers and computer vision experts, this perspective does not capture the business realities of the CAPTCHA-solving ecosystem. Arms races in computer security (e.g., anti-virus, anti-spam, etc.) traditionally favor the adversary, largely because the attacker's role is to generate new instances while the defender must recognize them—and the recognition problem is almost always much harder. However, CAPTCHAs reverse these roles since Web sites can be agile in their use of new CAPTCHA types, while attackers own the more challenging recognition problem. Thus, the economics of automated solving are driven by several factors: the cost to develop new solvers, the accuracy of these solvers and the responsiveness of the sites whose CAPTCHAs are attacked.

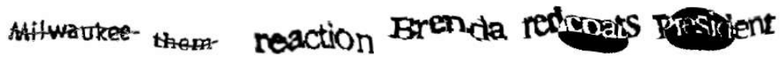
While it is difficult to precisely quantify the development cost for new solvers, it is clear that highly skilled labor is required and such developers must charge commensurate fees to recoup their time investment. Anecdotally, we contacted one such developer who was offering an automated solving library for the current reCaptcha CAPTCHA. He was charging $6,500 on a non-exclusive basis, and we did not pay to test this solver.

At the same time, as we saw with reCaptchaOCR, it can be particularly difficult to produce automated solvers that can deliver human-comparable accuracy (especially for "high-value" CAPTCHAs). While it seems that accuracy should be a minor factor since the cost of attempting a CAPTCHA is all but "free", in reality low success rates limit both the utility of a solver and its useful lifetime. In particular, over short time scales, many forums will blacklist an IP address after 5–7 failed attempts. More importantly, should a solver be put into wide use, changes in the gross CAPTCHA success rate over longer periods (e.g., days) is a strong indicator that a software solver is in use—a signature savvy sites use to revise their CAPTCHAs in turn.[5]

Thus, for a software solver to be profitable, its price must be less than the total value that can be extracted

[5]We are aware that some well-managed sites already have alternative CAPTCHAs ready for swift deployment in just such a situation.





(a) Early 2008 (b) December 16th 2009 (c) January 24th 2010

Figure 2: Examples of CAPTCHAs downloaded directly from reCaptcha at different time periods.

in the useful lifetime before the solver is detected and the CAPTCHA changed. Moreover, for this approach to be attractive, it must also cost less than the alternative: using a human CAPTCHA-solving service. To make this tradeoff concrete, consider the scenario in which a CAPTCHA-solving service provider must choose between commissioning a new software solver (e.g., for a variant of a popular CAPTCHA) or simply outsourcing recognition piecemeal to human laborers. If we suppose that it costs $10,000 to implement a solver for a new CAPTCHA type with a 30% accuracy (like reCaptchaOCR), then it would need to be used over 65 million times (20 million successful) before it was a better strategy than simply hiring labor at $0.5/1,000.[6] However, the evidence from reCaptcha's response to reCaptchaOCR suggests that CAPTCHA providers are well able to respond before such amortization is successful. Indeed, in our interview, MR. E said that he had dabbled with automated solving but that new solvers stopped working too quickly. In his own words, "It is a big waste of time."

For these reasons, software solvers appear to have been relegated to a niche status in the solving ecosystem—focusing on those CAPTCHAs that are static or change slowly in response to pressure. While a technological breakthrough could reverse this state of affairs, for now it appears that human-based solving has come to dominate the commercial market for service.

# 4 Human Solver Services

Since CAPTCHAs are only intended to obstruct automated solvers, their design point can be entirely sidestepped by outsourcing the task to human labor pools, either opportunistically or on a "for hire" basis. In this section, we review the evolution of this labor market, its basic economics and some of the underlying ethical issues that informed our subsequent measurement study.

## 4.1 Opportunistic Solving

Opportunistic human solving relies on convincing an individual to solve a CAPTCHA as part of some other unrelated task. For example, an adversary controlling access to a popular Web site might use its visitors to opportunistically solving third-party CAPTCHAs by offering these challenges as its own [1, 8]. A modern variant of this approach has recently been employed by the Koobface botnet, which asks infected users to solve a CAPTCHA (under the guise of a Microsoft system management task) [13]. However, we believe that retention of these unwitting solvers will be difficult due to the high profile nature and annoyance of such a strategy, and we do not believe that opportunistic solving plays a major role in the market today.

[6]Moreover, human labor is highly flexible and can be used for the wide variety of CAPTCHAs demanded by customers, while a software solver inevitably is specialized to one particular CAPTCHA type.

## 4.2 Paid Solving

Our focus is instead on paid labor, which we believe now represents the core of the CAPTCHA-solving ecosystem, and the business model that has emerged around it. Figure 3 illustrates a typical workflow and the business relationships involved.

The premise underlying this approach is that there exists a pool of workers who are willing to interactively solve CAPTCHAs in exchange for less money than the solutions are worth to the client paying for their services.

The earliest description we have found for such a relationship is in a Symantec Blog post from September 2006 that documents an advertisement for a full-time CAPTCHA solver [20]. The author estimates that the resulting bids were equivalent to roughly one cent per CAPTCHA solved, or $10/1,000 (solving prices are commonly expressed in units of 1,000 CAPTCHAs solved). Starting from this date, one can find increasing numbers of such advertisements on "work-for-hire" sites such as getafreelancer.com, freelancejobsearch.com, and mistersoft.com. Shortly thereafter, *retail* CAPTCHA-solving services began to surface to resell such capabilities to a broad range of customers.

Moreover, a fairly standard business model has emerged in which such retailers aggregate the *demand* for CAPTCHA-solving services via a public Web site and open API. The example in Figure 3 shows the DeCaptcher service performing this role in steps ② and ⑥. In addition, these retailers aggregate the *supply* of CAPTCHA-solving labor by actively recruiting individuals to participate in both public and private Web-based "job sites" that provide online payments for CAPTCHAs solved. PixProfit, a worker aggregator for the DeCaptcher service, performs this role in steps ③–⑤ in the example.

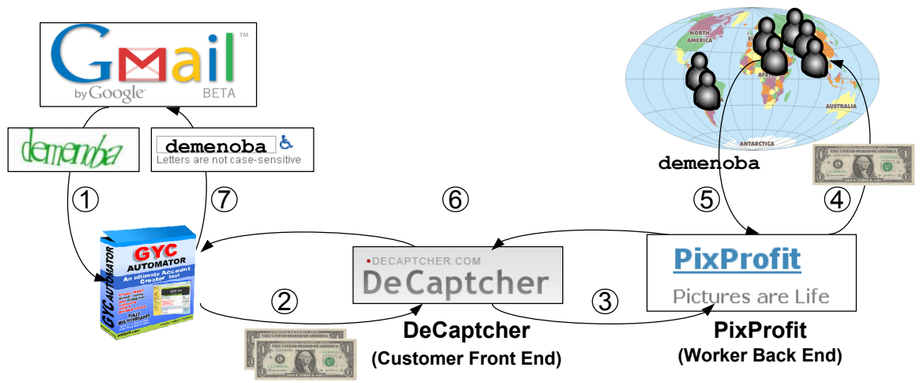



Figure 3: CAPTCHA-solving market workflow: ① GYC Automator attempts to register a Gmail account and is challenged with a Google CAPTCHA. ② GYC uses the DeCaptcher plug-in to solve the CAPTCHA at $2/1,000. ③ DeCaptcher queues the CAPTCHA for a worker on the affiliated PixProfit back end. ④ PixProfit selects a worker and pays at $1/1,000. ⑤ Worker enters a solution to PixProfit, which ⑥ returns it to the plug-in. ⑦ GYC then enters the solution for the CAPTCHA to Gmail to register the account.

### 4.3 Economics

While the market for CAPTCHA-solving services has expanded, the wages of workers solving CAPTCHAs have been declining. A cursory examination of historical advertisements on getafreelancer.com shows that, in 2007, CAPTCHA solving routinely commanded wages as high as $10/1,000, but by mid-2008 a typical offer had sunk to $1.5/1,000, $1/1,000 by mid-2009, and today $0.75/1,000 is common, with some workers earning as little as $0.5/1,000.

This downward price pressure reflects the commodity nature of CAPTCHA solving. Since solving is an unskilled activity, it can easily be sourced, via the Internet, from the most advantageous labor market—namely the one with the lowest labor cost. We see anecdotal evidence of precisely this pattern as advertisers switched from pursuing laborers in Eastern Europe to those in Bangladesh, China, India and Vietnam (observations further corroborated by our own experimental results later).

Moreover, competition on the retail side exerts pressure for all such employers to reduce their wages in turn. For example, here is an excerpt from a recent announcement at typethat.biz, the "worker side" of one such CAPTCHA-solving service:

```
009-12-14 13:54 Admin post
Hello, as you could see, server was unstable
last days. We can't get more captchas
because of too high prices in comparison
with other services. To solve this problem,
unfortunately we have to change the rate,
on Tuesday it will be reduced.
```

Shortly thereafter, typethat.biz reduced their offered rate from $1/1,000 to $0.75/1,000 to stay competitive.

These changes reflect similar decreases on the retail side: the customer cost to have 1,000 CAPTCHAs solved is now commonly $2/1,000 and can be as low as $1/1,000. To protect prices, a number of retailers have tried to tie their services to third-party products with varying degrees of success. For example, GYC Automator is a popular "black hat" bulk account creator for Gmail, Yahoo and Craigslist; Figure 3 shows GYC's role in the CAPTCHA ecosystem, with the tool scraping a CAPTCHA in step ① and supplying a CAPTCHA solution in step ⑦. GYC has a relationship with the CAPTCHA-solving service Image2Type (not to be confused with ImageToType). Similarly, SENuke is a blog and forum spamming product that has integral support for two "up-market" providers, BypassCaptcha and BeatCaptchas. In both cases, this relationship allows the CAPTCHA-solving services to charge higher rates: roughly $7/1,000 for BypassCaptcha and BeatCaptchas, and over $20/1,000 for Image2Type. It also provides an ongoing revenue source for the software developer. For his service, MR. E confirms that software partners bring in many customers (indeed, they are the majority revenue source) and that he offers a variety of revenue sharing options to attract such partners.

However, such large price differences encourage arbitrage, and in some cases third-party developers have created plug-ins to allow the use of cheaper services on such packages. Indeed, in the case of GYC Automator, an independent developer built a DeCaptcher plug-in which reduced the solving cost by over an order of magnitude. This development has created an ongoing conflict between the seller of GYC Automator and the distributor of the DeCaptcher plug-in. Other software developers have chosen to forgo large margin revenue sharing in favor of service diversity. For example, modern versions of the Xrumer package can use multiple price-leading services (Antigate and CaptchaBot).

Finally, while it is challenging to measure profitability directly, we have one anecdotal data point. In our discussions with MR. E, whose service is in the middle of the price spectrum, he indicated that routinely 50% of his revenue is profit, roughly 10% is for servers and bandwidth, and the remainder is split between solving labor and incentives for partners.

### 4.4 Active Measurement Issues

The remainder of our paper focuses on active measurement of such services, both by paying for solutions and by participating in the role of a CAPTCHA-solving laborer. The security community has become increasingly aware of the need to consider the legal and ethical context of its actions, particularly for such active involvement, and we briefly consider each in turn for this project.

In the United States (we restrict our brief discussion to U.S. law since that is where we operate), there are several bodies of law that may impinge on CAPTCHA solving. First, even though the services being protected are themselves "free", it can be argued that CAPTCHAs are an access control mechanism and thus evading them exceeds the authorization granted by the site owner, in potential violation of the Computer Fraud and Abuse Act (and certainly of their terms of service). While this interpretation is debatable, it is a moot point for our study since we never make use of solved CAPTCHAs and thus never access any of the sites in question. A trickier issue is raised by the Digital Millennium Copyright Act's anti-circumvention clause. While there are arguments that CAPTCHA solvers provide a real use outside circumvention of copyright controls (e.g., as aids for the visually impaired) it is not clear—especially in light of increasingly common audio CAPTCHA options—that such a defense is sufficient to protect infringers. Indeed, Ticketmaster recently won a default judgment against RMG Technologies (who sold automated software to bypass the Ticketmaster CAPTCHA) using just such an argument [2]. That said, while one could certainly apply the DMCA against those offering a *service* for CAPTCHA-solving purposes, it seems a stretch to include individual human workers as violators since any such "circumvention" would include innate human visual processes.

Aside from potential legal restrictions, there are also related ethical concerns; one can do harm without such actions being illegal. In considering these questions, we use a consequentialist approach – comparing the consequences of our intervention to an alternate world in which we took no action — and evaluate the outcome for its cost-benefit tradeoff.

On the purchasing side, we impart no direct impact since we do not actually *use* the solutions on their respective sites. We *do* have an indirect impact however since, through purchasing services, we are providing support to both workers and service providers. In weighing this risk, we concluded that the indirect harm of our relatively small investment was outweighed by the benefits that come from better understanding the nature of the threat. On the solving side, the ethical questions are murkier since we understand that solutions to such CAPTCHAs *will* be used to circumvent the sites they are associated with. To sidestep this concern, we chose *not* to solve these CAPTCHAs ourselves. Instead, for each CAPTCHA one of our worker agents was asked to solve, we proxied the image back into the same service via the associated retail interface. Since each CAPTCHA is then solved by the *same* set of solvers who would have solved it *anyway*, we argue that our activities do not impact the gross outcome. This approach does cause slightly more money to be injected into the system, but this amount is small.

Finally, we consulted with our human subjects liaison on this work and we were told that the study did not require approval.

## 5 Solver Service Quality

In this section we present our analysis of CAPTCHA-solving services based on actively engaging with a range of services as a client. We evaluate the customer interface, solution accuracy, response time, availability, and capacity of the eight retail CAPTCHA-solving services listed in Table 1.

We chose these services through a combination of Web searching and reading Web forums focused on "black-hat" search-engine optimization (SEO). In October of 2009, we selected the eight listed in Table 1 because they were well-advertised and reflected a spectrum of price offerings at the time. Over the course of our study, two of the services (CaptchaGateway and CaptchaBypass) ceased operation—we suspect because of competition from lower-priced vendors.

### 5.1 Customer Account Creation

For most of these services, account registration is accomplished via a combination of the Web and e-mail: contact information is provided via a Web site and subsequent sign-up interactions are conducted largely via e-mail. However, most services presented some obstacles

| Service | $/1K Bulk | Dates (2009–2010) | Requests | Responses |
|---|---|---|---|---|
| Antigate (AG) | $1.00 | Oct 06 – Feb 01 (118 days) | 28,210 | 27,726 (98.28%) |
| BeatCaptchas (BC) | $6.00 | Sep 21 – Feb 01 (133 days) | 28,303 | 25,708 (90.83%) |
| BypassCaptcha (BY) | $6.50 | Sep 23 – Feb 01 (131 days) | 28,117 | 27,729 (98.62%) |
| CaptchaBot (CB) | $1.00 | Oct 06 – Feb 01 (118 days) | 28,187 | 22,677 (80.45%) |
| CaptchaBypass (CP) | $5.00 | Sep 23 – Dec 23 (91 days) | 17,739 | 15,869 (89.46%) |
| CaptchaGateway (CG) | $6.60 | Oct 21 – Nov 03 (13 days) | 1,803 | 1,715 (95.12%) |
| DeCaptcher (DC) | $2.00 | Sep 21 – Feb 01 (133 days) | 28,284 | 24,411 (86.31%) |
| ImageToText (IT) | $20.00 | Oct 06 – Feb 01 (118 days) | 14,321 | 13,246 (92.49%) |

Table 1: Summary of the customer workload to the CAPTCHA-solving services.

to account creation, reflecting varying degrees of due diligence.

For example, both CaptchaBot and Antigate required third-party "invitation codes" to join their services, which we acquired from the previously mentioned forums. Interestingly, Antigate guards against Western users by requiring site visitors to enter the name of the Russian prime minister in Cyrillic before granting access—an innovation we refer to as a "culturally-restricted CAPTCHA".[7] Some services require a live phone call for account creation, for which we used an anonymous mobile phone to avoid any potential biases arising from using a University phone number. In our experience, however, the burden of proof demanded is quite low and our precautions were likely unnecessary. For example, setting up an ImageToText account required a validation call, but the only question asked was "Did you open an account on ImageToText?" Upon answering in the affirmative (in a voice clearly conflicting with the gender of the account holder's name), our account was promptly enabled. For one service, DeCaptcher, we created multiple accounts to evaluate whether per-customer rate limiting is in use (we found it was not).

Finally, each service typically requires prepayment by customers, in units defined by their price schedule (1,000 CAPTCHAs is the smallest "package" generally offered). To fund each account, we used prepaid VISA gift cards issued by a national bank unaffiliated with our university.

## 5.2 Customer Interface

Most services provide an API package for uploading CAPTCHAs and receiving results, often in multiple programming languages; we generally used the PHP-based APIs. BeatCaptchas and BypassCaptcha did not offer pre-built API packages, so we implemented our own API in Ruby to interface with their Web sites. The client APIs generally employ one of two methods when interacting with their corresponding services. In the first, the API client performs a single HTTP POST that uploads the image to the service, waits for the CAPTCHA to be solved, and receives the answer in the HTTP response; BeatCaptchas, BypassCaptcha, CaptchaBypass and CaptchaBot utilize this method.

In the second, the client performs one HTTP POST to upload the image, receives an image ID in the response, and subsequently polls the site for the CAPTCHA solution using the image ID; Antigate, CaptchaGateway, and ImageToText employ this approach. These APIs recommend poll rates between 1–5 seconds; we polled these services once per second. DeCaptcher uses a custom protocol that is not based on HTTP, although they also offer an HTTP interface. One interesting note about ImageToText is that customers must verify that their API code works in a test environment before gaining access to the actual service. The test environment allows users to see the CAPTCHAs they submit and solve them manually.

## 5.3 Service Pricing

Several of the services, notably Antigate and DeCaptcher, offer bidding systems whereby a customer can offer payment over the market rate in exchange for higher priority access to solvers when load is high. In our experience, DeCaptcher charges customers their full bid price, while Antigate typically charges at a lower rate depending on load (as might happen in a second-price auction). To effectively use Antigate, we set our bid price to $2/1,000 solutions since we experienced a large volume of load shedding error codes at the minimum bid price of $1/1,000 (Section 5.9 reports on our experiences with service load in more detail). We have not seen price fluctuations on the worker side of these services, and thus we believe that this overage represents pure profit to the service provider.

[7] In principle, such an approach could be used to artificially restrict labor markets to specific cultures (i.e., CAPTCHA labor protectionism). However it is an open problem if such a *general* form of culturally-restricted CAPTCHA can be devised that has both a large number of examples and a low false reject rate from its target population.



## 5.4 Test Corpus

We evaluated the eight CAPTCHA-solving services in Table 1 as a customer over the course of about five months using a representative sample of CAPTCHAs employed by popular Web sites. To collect this CAPTCHA workload, we assembled a list of 25 popular Web sites with unique CAPTCHAs based on the Alexa rank of the site and our informal assessment of its value as a target (see Figure 5 for the complete list). We also used CAPTCHAs from reCaptcha, a popular CAPTCHA provider used by many sites. We then collected about 7,500 instances of each CAPTCHA directly from each site. For the capacity measurement experiments (Section 5.8), we used 12,000 instances of the Yahoo CAPTCHA graciously provided to us by Yahoo.

## 5.5 Verifying Solutions

To assess the accuracy of each service, we needed to determine the correct solution for each CAPTCHA in our corpus. We used the services themselves to do this for us. For each instance, we used the most frequent solution returned by the solver services, after normalizing capitalization and whitespace. If there was more than one most frequent solution, we treated all answers as incorrect (taking this to mean that the CAPTCHA had no correct solution). Table 1 shows the overall accuracy of each service as given by our method.

To validate this heuristic, we randomly selected 1,025 CAPTCHAs having at least one service-provided solution and manually examined the images. Of these, we were able to solve 1,009, of which 940 had a unique plurality that agreed with our solution, giving an error rate for the heuristic of just over 8%. Of the 16 CAPTCHAs (1.6%) we could not solve, seven were entirely unreadable, six had ambiguous characters (e.g., '0' vs. 'o', '6' vs. 'b'), and three were rendered ambiguous due to overlapping characters. (We note that Bursztein *et al.* [3] removed CAPTCHAs with no majority from their calculation, which resulted in a higher estimated accuracy than we found in our study.)

## 5.6 Quality of Service

To assess the accuracy, response time, and service availability of the eight CAPTCHA solving services, we continuously submitted CAPTCHAs from our corpus to each service over the course of the study. We submitted a single CAPTCHA every five minutes to all services simultaneously, recording the time when we submitted the CAPTCHA and the time when we received the response. Recall that ImageToText, Antigate and CaptchaGateway require customers to poll the service for the response to a submitted CAPTCHA; we paused one second between each poll call.

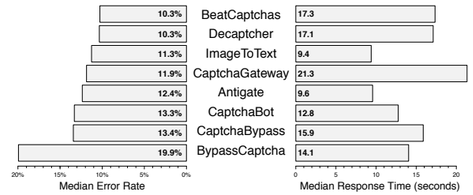



Figure 4: Median error rate and response time (in seconds) for all services. Services are ranked top-to-bottom in order of increasing error rate.

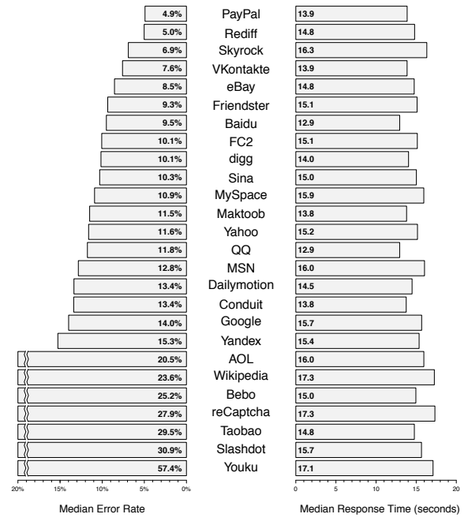



Figure 6: Median error rate and response time (in seconds) for all CAPTCHAs. CAPTCHAs are ranked top-to-bottom in order of increasing error rate.

Table 1 also summarizes the dates, durations, and number of CAPTCHA requests we submitted to the services; Figure 5 presents the error rate and mean response time at a glance for each combination of solver service and CAPTCHA type. We used each service for up to 118 days, submitting up to 28,303 requests per service during that period. We were not able to submit the same number of CAPTCHAs to all services for a number of reasons. For example, services would go offline temporarily, or we would rewrite parts of our client implementation, thus requiring us to temporarily remove the service from the experiment. Furthermore, CaptchaGateway and CaptchaBypass ceased operation during our study.

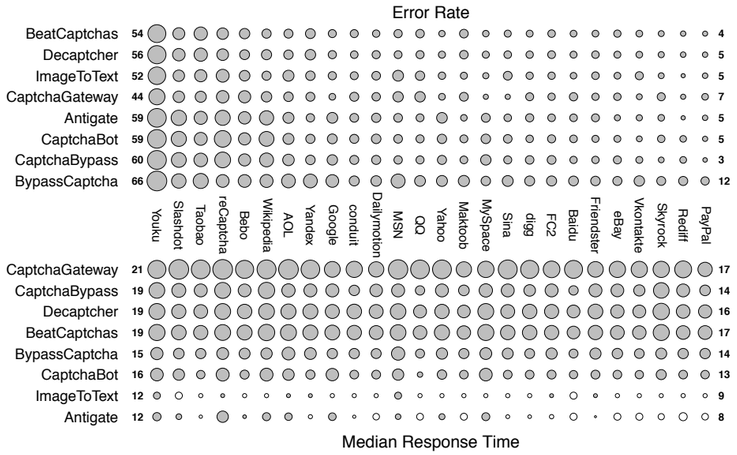



Figure 5: Error rate and median response time for each combination of service and CAPTCHA type. The area of each circle upper table is proportional to the error rate (among solved CAPTCHAs). In the lower table, circle area is proportional to the response time *minus ten seconds* (for increased contrast); negative values are denoted by unshaded circles. Numeric values corresponding to the values in the leftmost and rightmost columns are shown on the side. Thus, the error rate of BypassCaptcha on Youku CAPTCHAs is 66%, and for BeatCaptchas on PayPal 4%. The median response time of CaptchaGateway on Youku is 21 seconds, and 8 seconds for Antigate on PayPal.

### Accuracy

A CAPTCHA solution is only useful if it is correct. The left bar plot in Figure 4 shows the median error rate for each service. Overall the services are reasonably accurate: with the exception of BypassCaptcha, 86–89% of responses [8] were correct. This level of accuracy is in line with results reported by Bursztein *et al.* [3] for human solvers and substantially better than the accuracy of reCaptchaOCR (Section 3).

By design, CAPTCHAs vary in difficulty. Do the observed error rates reflect such differences? The top half of Figure 5 shows service accuracy (in terms of its error rate) on each CAPTCHA type. The area of each circle is proportional to a service's mean error rate on a particular CAPTCHA type. Services are arranged along the $y$-axis in order of increasing accuracy, with the most accurate (lowest error rate) at the top and the least accurate (highest error rate) at the bottom. CAPTCHA types are arranged in decreasing order of their median error rate. The median error rate of each type is also shown in Figure 6.

Accuracy clearly depends on the type of CAPTCHA. The error rate for ImageToText with Youku, for instance, is 5 times its PayPal error rate. Furthermore, the ranking of CAPTCHA accuracies are generally consistent across the services—all services have relatively poor accuracy on Youku and good accuracy on PayPal.

Based on the data, one might conclude that a group of CAPTCHAs on the left headed by Youku, reCaptcha, Slashdot, and Taobao are "harder" than the rest. However an important factor affecting solution accuracy (as well as response time) in our measurements is worker familiarity with a CAPTCHA type. In the case of Youku, for instance, workers may simply be unfamiliar with these CAPTCHAs. On the other hand, workers are likely familiar with reCaptcha CAPTCHAs (see Section 6.6), which may genuinely be "harder" than the rest. As a point of comparison, MR. E reported in our interview that his service experiences a 5–10% error rate. Since his CAPTCHA mix is likely different, and less diverse, than our full set, his claim seems reasonable.

### Response Time

In addition to accuracy, customers want services that solve CAPTCHAs quickly. Figure 7 shows the cumulative distribution of response times of each service. The curves of CaptchaBot, CaptchaBypass, ImageToText, and Antigate exhibit the quantization effect of polling—either in the client API or on the server—as a stair-step pattern. The shape of the distributions is characteristically log-normal, with a median response of 14 seconds (across all services) and a third-quartile response time of 20 seconds—well within the session timeout of most Web sites. For convenience, Figure 4 also shows median response times for each service. In contrast to Bursztein *et al.* [3], who used a different labor pool (Amazon Mechanical Turk), we found no significant difference in response times of correct and incorrect responses.

Services differ considerably in the relative response times they provide to their customers. Antigate (for which we paid a slight premium for priority service as described in Section 5.3) and ImageToText provided the fastest service with median response times of 9.6 seconds and 9.4 seconds, respectively, with 90% of CAPTCHAs solved under 25 seconds. CaptchaGateway was the slowest service we measured, with a median of 21.3 seconds and 10% of responses taking over a minute; it was also one of the two services that ceased operation during our study. The remaining services fall in between those extremes. MR. E reported that his service trains workers to achieve response times of 10–12 seconds on average, which is consistent with our measurements of his service.

DeCaptcher and BeatCaptchas have very similar distributions. We have seen evidence (i.e., error messages from BeatCaptchas that are identical to ones documented for the DeCaptcher API) that suggests that BeatCaptchas uses DeCaptcher as a back end. Antigate returns some correct responses unusually quickly (a few seconds), for which we currently do not have an explanation; we have ruled out caching effects.

Services have an advantage if they have better response times than their competition, and the services we measured differ substantially. We suspect that it is a combination of two factors: software and queueing delay in the service infrastructure, and worker efficiency. Antigate, for instance, appears to have an unusually large labor pool (Section 5.8), which may enable them to keep queueing delay low. Similarly, ImageToText appears to have an adaptive, high-quality labor pool (Section 6.4). We observed additional delays of 5 seconds due to load (Section 5.9), but load likely affects all services similarly.

We found that accuracy varied with the type of CAPTCHA. A closely related issue is to what degree response time also varies according to CAPTCHA type. The bottom of Figure 5 shows response times by CAPTCHA type. Services are listed along the $y$-axis from slowest (top) to fastest service (bottom). The area of each circle is proportional to the median response time of a service on a particular CAPTCHA type *minus ten seconds* (for greater contrast). Shaded circles are times in excess of ten seconds, unshaded circles are times less than ten seconds. For example, the median response time of Antigate on PayPal CAPTCHAs—8 seconds—is shown as an unshaded circle. Note that CAPTCHA types are still sorted by *accuracy*. The right half of Figure 4 aggregates response times by service, showing the median response time of each.

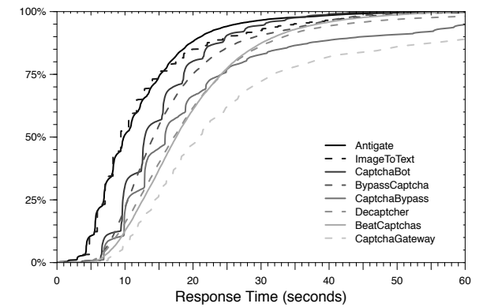



Figure 7: Cumulative distribution of response times for each service.

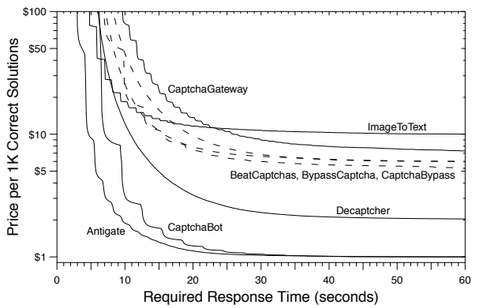



Figure 8: Price for 1,000 correctly-solved CAPTCHAs within a given response time threshold.

We see some variation in response time among CAPTCHA types. Youku and reCaptcha, for instance, consistently induce longer response times across services, whereas Baidu, eBay, and QQ consistently have shorter response times. However, the variation in response times among the services dominates the variation due to CAPTCHA type. The fastest CAPTCHAs that DeCaptcher solves (e.g., Baidu and QQ) are slower on average than the slowest CAPTCHAs that Antigate and ImageToText solve.

## 5.7 Value

CAPTCHA solvers differ in terms of accuracy, response time, and price. The *value* of a particular solver to a customer depends upon the combination of all of these factors: a customer wants to pay the lowest price for both fast and accurate CAPTCHAs. For example, suppose that a customer wants to create 1,000 accounts on an Internet service, and the Internet service requires that CAPTCHAs be solved within 30 seconds. When using a CAPTCHA solver, the customer will have to pay to have at least 1,000 CAPTCHAs solved, and likely more due to solutions with response times longer than the 30-second

[8] The error rate is over received responses and does not include rejected requests. We consider response rate to be a measure of *availability* rather than accuracy.

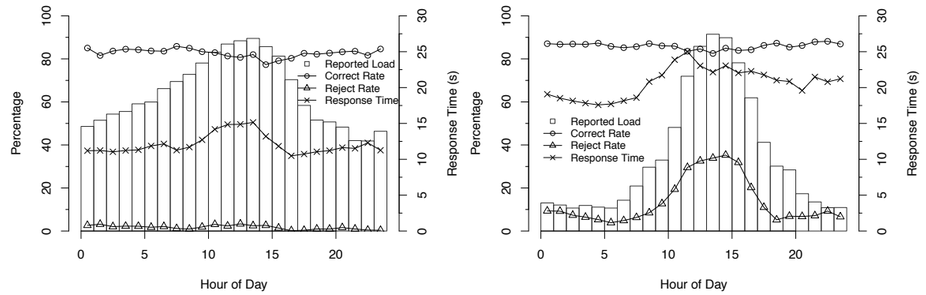



Figure 9: Load reported by (a) Antigate and (b) DeCaptcher as a function of time-of-day in one-hour increments. For comparison, we show the percentage of correct responses and rejected requests per hour, as well as the average response time per hour.

threshold (recall that customers do not have to pay for incorrect solutions). From this perspective, the solver with the best value may not be the one with the cheapest price.

Figure 8 explores the relationship among accuracy, response time, and price for this scenario. The $x$-axis is the time threshold $T$ within which a CAPTCHA is useful to a customer. The $y$-axis is the *adjusted price* per bundle of 1,000 CAPTCHAs that are both solved correctly *and* solved within time $T$. Each curve corresponds to a solver. Each solver charges a price per CAPTCHA solved (Table 1), but not all solved CAPTCHAs will be useful to the customer. The adjusted price therefore includes the overhead of solving CAPTCHAs that take longer than $T$ and are effectively useless. Consider an example where a customer wants to have 1,000 correct CAPTCHAs solved within 30 seconds, a solver charges \$2/1,000 CAPTCHAs, and 70% of the solver's CAPTCHA responses are correct and returned within 30 seconds. In this case, the customer will effectively pay an adjusted price of $\$2 \times (1/0.70) = \$2.86/1,000$ useful CAPTCHAs.

The results in Figure 8 show that the solver with the best value depends on the response time threshold. For high thresholds (more than 25 seconds), both Antigate and CaptchaBot provide the best value and ImageToText is the most expensive as suggested by their bulk prices (Table 1). However, below this threshold the rankings begin to change. Antigate begins to have better value than CaptchaBot due to having consistently better response times. In addition, ImageToText starts to overtake the other services. Even though its bulk price is $5x$ that of DeCaptcher, for instance, its service is a better value for having CAPTCHAs solved within 8 seconds (albeit at a premium adjusted price).

### 5.8 Capacity

Another point of differentiation is solver capacity, namely how many CAPTCHAs a service can solve in a given unit of time. In addition to low-rate measurements, we also attempted to measure a service's maximum capacity using bursts of CAPTCHA requests. Specifically, we measured the number and rate of solutions returned in response to a given offered load, substantially increasing the load in increments until the service appeared overloaded. We carried out this experiment successfully for five of the services. Of them, Antigate had by far the highest capacity, solving on the order of 27 to 41 CAPTCHAs per second. Even at our highest sustained offered load (1,536 threads submitting CAPTCHAs simultaneously, bid set at \$3/1,000), our rejection rate was very low, suggesting that Antigate's actual capacity may in fact be higher. Due to financial considerations, we did not attempt higher offered loads.

For the remaining services, we exceeded their available capacity. We took a non-negligible reject rate to be an indicator of the service running at full capacity. Both DeCaptcher and CaptchaBot were able to sustain a rate of about 14–15 CAPTCHAs per second, with BeatCaptchas and BypassCaptchas sustaining a solve rate of eight and four CAPTCHAs per second, respectively.

Based on these rates, we can calculate a rough estimate of the number of workers at these services. Assuming 10–13 seconds per CAPTCHA (based on our interview with MR. E, and consistent with our measured latencies of his service in the 10–20 second range), Antigate would have had at least 400–500 workers available to service our request. Since we did not exceed their available capacity, the actual number may be larger. Both DeCaptcher and CaptchaBot, at a solve rate of 15 CAPTCHAs per second mentioned above, would have had 130–200 workers available.

### 5.9 Load and Availability

Customers can poll the transient load on the services and offer payment over the market rate in exchange for higher priority access when load is high. During our background CAPTCHA data collection for these services, we also recorded the transient load that they reported. From these measurements, we can examine to what extent services report substantial load, and correlate reported load with other observable metrics (response time, reject rate) to evaluate the validity of the load reports. Because DeCaptcher charges the full customer bid independent of actual load, for instance, it might be motivated to report a false high load in an attempt to encourage higher bids from customers.

Figure 9 shows the average reported load as a function of the time of day (in the US Pacific time zone) for both services: for each hour, we compute the average of all load samples taken during that hour for all days of our data set. Antigate reports a higher nominal background load than DeCaptcher, but both services clearly report a pronounced diurnal load effect.

For comparison, we also overlay three other service metrics for each hour across all days: average response time of solved CAPTCHAs, percentage of submitted CAPTCHAs rejected by the service, and the percentage of responses with correct solutions. Response time correlates with reported load, increasing by 5 seconds during high load for each service—suggesting that the high load reports are indeed valid. The percentage of rejected requests for DeCaptcher further validates the load reports. When our bids to DeCaptcher were at the base price of \$2/1,000 at times of high load, DeCaptcher aggressively rejected our work requests. To confirm that a higher bid resulted in lower rejection rates, we measured available capacity at 5PM (US Pacific time) at the base price of \$2 and then, a few minutes later, at \$5, obtaining solve rates of 8 and 18 CAPTCHAs per second, respectively. Although not conclusive, this experience suggests that higher bids may be necessary to achieve a desired level of service at times of high load. Likewise, Antigate exhibits better quality of service when bidding \$1 over the base price, though bidding over this amount produced no noticeable improvement (we tested up to \$6/1,000).

As further evidence, recall that for Antigate we had to offer premium bids before the service would solve our requests (Section 5.2). As a result, even during high loads Antigate did not reject our requests, presumably prioritizing our requests over others with lower bids.

Finally, as expected, accuracy is independent of load: workers are shielded from load behind work queues, solving CAPTCHAs to their ability unaffected by the offered load on the system.

## 6 Workforce

Human CAPTCHA solving services are effectively aggregators. On one hand, they aggregate demand by providing a singular point for purchasing solving services. At the same time, they aggregate the labor supply by providing a singular point through which workers can depend on being offered consistent CAPTCHA solving work for hire. Thus, for each of the publicly-facing retail sites described previously, there is typically also a private "job site" accessed by workers to receive CAPTCHA images and provide textual solutions. Identifying these job sites and which retail service they support is an investigative challenge. For this study, we focused our efforts on two services for which we feel confident about the mapping: Kolotibablo and PixProfit. Kolotibablo is a Russian-run job site that supplies solutions for the retail service Antigate (which, along with CaptchaBot, is the current price leader).


Figure 10: Portion of a PixProfit worker interface displaying a Microsoft CAPTCHA.

### 6.1 Account Creation

For each job site, account creation is similar to the retail side, but due diligence remains minimal. As a form of quality control, some job sites will evaluate new workers using a corpus of "test" CAPTCHAs (whose solutions are known *a priori*) before they allow them to solve externally provided CAPTCHAs. For this reason, we discard the first 30 CAPTCHAs provided by PixProfit, which we learned by experience correspond to test CAPTCHAs.

### 6.2 Worker Interface

Services provide workers with a Web based interface that, after logging in, displays CAPTCHAs to be solved and provides a text box for entering the solution (Figure 10 shows an example of the interface for PixProfit). Each site also tracks the number of CAPTCHAs solved, the number that were reported as correct (by customers of the retail service), and the amount of money earned. PixProfit also assigns each worker a "priority" based on solution accuracy. Better accuracy results in more CAPTCHAs to solve during times of lower load. If a solver's accuracy decreases too much, services ban the account. In our experiments, our worker agents always used fresh accounts with the highest level of priority.

| Language | Example | AG | BC | BY | CB | DC | IT | All |
|---|---|---|---|---|---|---|---|---|
| English | one two three | 51.1 | 37.6 | 4.76 | 40.6 | 39.0 | 62.0 | 39.2 |
| Chinese (Simp.) | 一 二 三 | 48.4 | 31.0 | 0.00 | 68.9 | 26.9 | 35.8 | 35.2 |
| Chinese (Trad.) | 一 二 三 | 52.9 | 24.4 | 0.00 | 63.8 | 30.2 | 33.0 | 34.1 |
| Spanish | uno dos tres | 1.81 | 13.8 | 0.00 | 2.90 | 7.78 | 56.8 | 13.9 |
| Italian | uno due tre | 3.65 | 8.45 | 0.00 | 4.65 | 5.44 | 57.1 | 13.2 |
| Tagalog | isá dalawá tatló | 0.00 | 5.79 | 0.00 | 0.00 | 7.84 | 57.2 | 11.8 |
| Portuguese | um dois três | 3.15 | 10.1 | 0.00 | 1.48 | 3.98 | 48.9 | 11.3 |
| Russian | один два три | 24.1 | 0.00 | 0.00 | 11.4 | 0.55 | 16.5 | 8.76 |
| Tamil | ஒன்று இரண்டு மூன்று | 2.26 | 21.1 | 3.26 | 0.74 | 12.1 | 5.36 | 7.47 |
| Dutch | een twee drie | 4.09 | 1.36 | 0.00 | 0.00 | 1.22 | 31.1 | 6.30 |
| Hindi | एक दो तीन | 10.5 | 5.38 | 2.47 | 1.52 | 6.30 | 9.49 | 5.94 |
| German | eins zwei drei | 3.62 | 0.72 | 0.00 | 1.46 | 0.58 | 29.1 | 5.91 |
| Malay | satu dua tiga | 0.00 | 1.42 | 0.00 | 0.00 | 0.55 | 29.4 | 5.23 |
| Vietnamese | một hai ba | 0.46 | 2.07 | 0.00 | 0.00 | 1.74 | 18.1 | 3.72 |
| Korean | 일 이 삼 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 20.2 | 3.37 |
| Greek | ένα δύο τρία | 0.45 | 0.00 | 0.00 | 0.00 | 0.00 | 15.5 | 2.65 |
| Arabic | واحد اثنين ثلاثة | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 15.3 | 2.56 |
| Bengali | এক দুই তিন | 0.45 | 0.00 | 9.89 | 0.00 | 0.00 | 0.00 | 1.72 |
| Kannada | ಒಂದು ಎರಡು ಮೂರು | 0.91 | 0.00 | 0.00 | 0.00 | 0.55 | 6.14 | 1.26 |
| Klingon | | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1.12 | 0.19 |
| Farsi | یک دو سه | 0.45 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.08 |

Table 2: Percentage of responses from the services with correct answers for the language CAPTCHAs.

### 6.3 Worker Wages

Kolotibablo pays workers at a variable rate depending on how many CAPTCHAs they have solved. This rate varies from $0.50/1,000 up to over $0.75/1,000 CAPTCHAs. PixProfit is the equivalent supplier for DeCaptcher and offers a somewhat higher rate of $1/1,000. Typically, workers must earn a minimum amount of money before payout ($3.00 at PixProfit and $1.00 at Kolotibablo), and services commonly provide payment via an online e-currency system such as WebMoney.

While we cannot directly measure the gross wages paid by either service, Kolotibablo provides a public list to its workers detailing the monthly earnings for the top 100 solvers each day (presumably as a worker incentive). We monitored these earnings for two months beginning on Dec. 1st, 2009. On this date, the average monthly payout among the top 100 workers was $106.31. However, during December, Kolotibablo revised its bonus payout system, which reduced the payout range by approximately 50% (again reflecting downward price pressure on CAPTCHA-solving labor). As a result, one month later on Jan. 1st, 2010, the average monthly payout to the top 100 earners decreased to $47.32. In general, these earnings are roughly consistent with wages paid to low-income textile workers in Asia [12], suggesting that CAPTCHA-solving is being outsourced to similar labor pools; we investigate this question next.

### 6.4 Geolocating Workers

We crafted CAPTCHAs whose solutions would reveal information about the geographic demographics of the CAPTCHA solvers. We created CAPTCHAs using words corresponding to digits in the native script of various languages (“uno”, “dos”, “tres”, etc., for the CAPTCHA challenge in Spanish), where the correct solution is the sequence of Roman numerals corresponding to those words (“1”, “2”, “3”, etc.) for any CAPTCHA in any language. Ideally, such CAPTCHAs should be easy to grasp and fast to solve by the language’s speakers, yet substantially less likely to be solved by non-speakers or random chance. We expect a measurably high accuracy for services employing workers familiar with those languages.

Table 2 lists the languages we used in this experiment along with an example three-digit CAPTCHA in the language corresponding to the solution “123”. For broad global coverage, we selected 21 languages based on a combination of factors including global exposure (English), prevalence of world-wide native speakers (Chinese, Spanish, English, Hindi, Arabic), regions of expected low-cost labor markets with inexpensive Internet access (India, China, Southeast Asia, Latin America), and developed regions unlikely to be sources of affordable CAPTCHA labor (e.g., Western Europe) and lastly one synthetic language as a control (Klingon [15]).

The CAPTCHA we submitted had instructions in the language for how to solve the CAPTCHA (e.g., “Por favor escriba los números abajo” for Spanish), as well as an initial word and Roman numeral as a concrete example (“uno”, “1”). In our experiments, we randomly generated 222 unique CAPTCHAs in each language and submitted them to the six services still operating in January 2010. We rotated through languages such that we submitted a CAPTCHA in this format once every 20–25 minutes. The CAPTCHAs did not repeat digits to reduce the correlated effect of a random guess. As a result, the actual probability for guessing a CAPTCHA is 1/504 ($9 \times 8 \times 7$, reduced by 1 due to the example), although workers unaware of the construction would still be making guesses out of 1,000 possibilities.

Table 2 also shows the accuracy of the services when presented with these CAPTCHAs. The accuracy corresponds to a response with all three digits correct (since we created them we have their ground truth). For a convenient ordering, we sort the languages by the average accuracy across all services.

The results paint a revealing picture. First, although Roman alphanumerics in typical CAPTCHAs are globally comprehensible—and therefore easily outsourced—English words for numerals represent a noticeable semantic gap for presumably non-English speakers. Very high accuracies on normal CAPTCHAs drop to 38–62% for the challenge presented in English.

Second, workers at a number of the services exhibit strong affinities to particular languages. Five of the services have accuracies for Chinese (Traditional and Simplified) either substantially higher or nearly as high as English. The services evidently include a sizeable workforce fluent in Chinese, likely mainland China with available low-cost labor. In addition, Antigate has appreciable accuracies for Russian and Hindi, presumably drawing on workforces in Russia and India. Similarly for CaptchaBypass and Russian; BeatCaptcha and Tamil, Portuguese, and Spanish; and DeCaptcher and Tamil. Other non-trivial accuracies in Bengali and Tagalog suggest further recruitment in India and southeast Asia. Services with non-trivial accuracies in Portuguese, Spanish, and Italian could be explained by a workforce familiar with one language who can readily deduce similar words in the other Romance languages. Consistent with these observations, MR. E reported in our interview that they draw from labor markets in China, India, Bangladesh, and Vietnam.


Figure 11: Custom Asirra CAPTCHA: workers must type the letters corresponding to pictures of cats.

Finally, the results for ImageToText are impressive. Relative to the other services, ImageToText has appreciable accuracy across a remarkable range of languages, including languages where none of the other services had few if any correct solutions (Dutch, Korean, Vietnamese, Greek, Arabic) and even two correct solutions of CAPTCHAs in Klingon. Either ImageToText recruits a truly international workforce, or the workers were able to identify the CAPTCHA construction and learn the correct answers. ImageToText is the most expensive service by a wide margin, but clearly has a dynamic and adaptive labor pool.

**Time Zone.** As another approach for using CAPTCHAs to reveal demographic information about workers—in this case, their time zone—we translated the following instruction into 14 of the languages as CAPTCHA images: “Enter the current time”. We sent these CAPTCHAs to each of the six services at the same rate as the other language CAPTCHAs with numbers. We received 15,775 responses, with the most common response being a retype of the instruction in the native language. Of the remaining responses, we received 1,583 (10.0%) with an answer in a recognizable time format. Of those, 77.9% of them came from UTC+8, further reinforcing the estimation of a large labor pool from China; the two other top time zones were the Indian UTC+5.5 with 5.7% and Eastern Europe UTC+2 with 3.0%.

### 6.5 Adaptability

As a final assessment, we wanted to examine how both CAPTCHA services and solvers adapt to changes in state-of-the-art CAPTCHA generation. We focused on the recently proposed Asirra CAPTCHA [9], which is based on identifying pictures of cats and dogs among a set of 12 images. Using the corpus of images provided by the Asirra authors, we hand crafted our own version of the

| Kolotibablo (Antigate) | | | | PixProfit (DeCaptcher) | | | |
|---|---|---|---|---|---|---|---|
| **Service** | **# CAPTCHAs** | **% Total** | **% Cum.** | **Service** | **# CAPTCHAs** | **% Total** | **% Cum.** |
| Microsoft | 6,552 | 25.5% | 25.5% | Microsoft | 12,135 | 43.1% | 43.1% |
| Vkontakte.ru | 5,908 | 23.0% | 48.5% | reCaptcha | 10,788 | 38.3% | 81.4% |
| Mail.ru | 3,607 | 14.0% | 62.5% | Google | 1,202 | 4.3% | 85.7% |
| Captcha.ru | 2,476 | 9.6% | 72.2% | Yahoo | 1,307 | 3.7% | 89.3% |
| reCaptcha | 921 | 3.6% | 75.8% | AOL | 415 | 1.5% | 90.8% |
| Other (18 sites) | 3680 | 14.3% | 90.1% | Other (18 sites) | 1086 | 3.9% | 94.7% |
| Unknown | 2551 | 9.9% | 100% | Unknown | 1505 | 5.3% | 100% |
| Total | 25,695 | | | Total | 28,166 | | |

Table 3: The top 5 targeted CAPTCHA types on Kolotibablo and PixProfit, based on CAPTCHAs observed posing as workers.

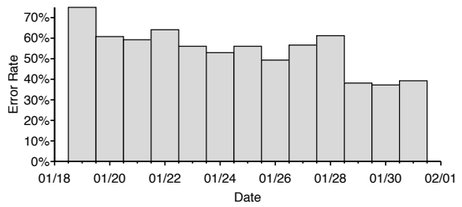



Figure 12: ImageToText error rate for the custom Asirra CAPTCHA over time.

CAPTCHA suitable for use with standard solver image APIs. Figure 11 shows an example. We wrote the instructions "Find all cats" in English, Chinese (Simpl.), Russian and Hindi across the top, as the majority of the workers speak one of these languages. We submitted this image once every three minutes to all services over 12 days. ImageToText displayed a remarkable adaptability to this new CAPTCHA type, successfully solving the CAPTCHA on average 39.9% of the time. Figure 12 shows the declining error rate for ImageToText; as time progresses, the workers become increasingly adept at solving the CAPTCHA. The next closest service was BeatCaptchas, which succeeded 20.4% of the time. The remaining services, excluding DeCaptcher, had success rates below 7%.

Coincidentally, as we were evaluating our own version of the Asirra CAPTCHA, on January 17th, 2010 DeCaptcher began offering an API method that supported it directly—albeit at $4 per 1,000 Asirra solves (double its base price). Microsoft had deployed the Asirra CAPTCHA on December 8th, 2009 on Club Bing. Demand for solving this CAPTCHA was apparently sufficiently strong enough that DeCaptcher took only five weeks to incorporate it into their service. We then performed the same experiment described above using the new DeCaptcher API method and received 1,494 responses. DeCaptcher successfully solved 696 (46.5%) requests with a median response time of 39 seconds, about 2.3 times its median of 17 seconds for regular CAPTCHAs. DeCaptcher appears to have factored in the longer solve times for the Asirra CAPTCHAs into the charged price. From what we can tell, though, DeCaptcher does not pay PixProfit workers double the amount for solving them, consequently increasing its profit margin on these new CAPTCHAs.

## 6.6 Targeted Sites

Customers of CAPTCHA-solving services target a number of different Web sites. Using our worker accounts on Kolotibablo and PixProfit, the public worker sites of Antigate and DeCaptcher, respectively, we can identify which Web sites are targeted by the customers of these services. Over the course of 82 days we recorded over 25,000 CAPTCHAs from Kolotibablo and 28,000 CAPTCHAs from PixProfit.

To identify the Web sites from which these CAPTCHAs originated, we first grouped the CAPTCHAs by image dimensions. Most groups consisted of a single CAPTCHA type, which we confirmed visually. We then attempted to identify the Web sites from which these CAPTCHAs were taken. In this manner we identified 90% of Kolotibablo CAPTCHAs and 94% of PixProfit CAPTCHAs.

Table 3 shows the top five CAPTCHA types we observed on Kolotibablo and PixProfit, with the remaining identified CAPTCHA types (18 CAPTCHA in both cases) representing 14% and 4% of the CAPTCHA volume on Kolotibablo and PixProfit respectively. Both distributions of CAPTCHA types are highly skewed: on PixProfit, the top two CAPTCHAs types represent 81% of the volume, with the top five accounting for 91%. Kolotibablo is not quite as concentrated, but the top five still account for 76% of its volume.

Clearly the markets for the services are different. Although Microsoft is by far the most common target for both, PixProfit tailors to CAPTCHAs from large global services (Google, Yahoo, AOL, and MySpace) whereas Russian sites otherwise dominate Kolotibablo (VKontakte.ru, Mail.ru, CAPTCHA.ru, Mamba.ru, and Yandex) — a demographic that correlates well with the observed worker fluency in Russian for Antigate (Table 2).



# 7 Discussion and Conclusion

By design, CAPTCHAs are simple and easy to solve by humans. Their "low-impact" quality makes them attractive to site operators who are wary of any defense that could turn away visitors. However, this same quality has made them easy to outsource to the global unskilled labor market. In this study, we have shed light on the business of solving CAPTCHAs, showing it to be a well-developed, highly-competitive industry with the capacity to solve on the order of a million CAPTCHAs per day. Wholesale and retail prices continue to decline, suggesting that this is a demand-limited market; an assertion further supported by our informal survey of several freelancer forums where workers in search of CAPTCHA-solving work greatly outnumber CAPTCHA-solving service recruitments. One may well ask: *Do* CAPTCHAs *actually work?* The answer depends on what it is that we expect CAPTCHAs to do.

**Telling computers and humans apart.** The original purpose of CAPTCHAs is to distinguish humans from machines. To this day, no completely general means of solving CAPTCHAs has emerged, nor is the cat-and-mouse game of creating automated solvers viable as a business model. In this regard, then, CAPTCHAs have succeeded.

**Preventing automated site access.** Today, the retail price for solving one million CAPTCHAs is as low as $1,000. Indeed, for well-motivated adversaries, CAPTCHAs are an acceptable cost of doing business when measured against the value of gaining access to the protected resource. E-mail spammers, for example, solve CAPTCHAs to gain access to Web mail accounts from which to send their advertisements, while blog spammers seek to acquire organic "clicks" and influence result placement on major search engines. Thus, in an absolute sense, CAPTCHAs do not prevent large-scale automated site access.

**Limiting automated site access.** However, it is shortsighted to evaluate CAPTCHAs as a defense in isolation. Rather, they exert friction on the underlying economic model and should be evaluated in terms of how efficiently they can undermine the attacker's profitability.

Put simply, a CAPTCHA reduces an attacker's expected profit by the cost of solving the CAPTCHA. If the attacker's revenue cannot cover this cost, CAPTCHAs as a defense mechanism have succeeded. Indeed, for many sites (e.g., low PageRank blogs), CAPTCHAs alone may be sufficient to dissuade abuse. For higher-value sites, CAPTCHAs place a utilization constraint on otherwise "free" resources, below which it makes no sense to target them. Taking e-mail spam as an example, let us suppose that each newly registered Web mail account can send some number of spam messages before being shut down. The marginal revenue per message is given by the average revenue per sale divided by the expected number of messages needed to generate a single sale. For pharmaceutical spam, Kanich *et al.* [14] estimate the marginal revenue per message to be roughly $0.00001; at $1 per 1,000 CAPTCHAs, a new Web mail account starts to break even only after about 100 messages sent.[9]

Thus, CAPTCHAs naturally limit site access to those attackers whose business models are efficient enough to be profitable in spite of these costs and act as a drag on profit for all actors. Indeed, MR. E reported that while his service had thousands of customers, 75% of traffic was generated by a small subset of them (5–10).

**The role of CAPTCHAs today.** Continuing our reasoning, the profitability of any particular scam is a function of three factors: the cost of CAPTCHA-solving, the effectiveness of any secondary defenses (e.g., SMS validation, account shutdowns, additional CAPTCHA screens, etc.) and the efficiency of the attacker's business model. As the cost of CAPTCHA solving decreases, a site operator must employ secondary defenses more aggressively to maintain a given level of fraud.

Unfortunately, secondary defenses are invariably more expensive both in infrastructure and customer impact when compared to CAPTCHAs. However, a key observation is that secondary defenses need only be deployed quickly enough to undermine profitability (e.g., within a certain number of messages sent, accounts registered per IP, etc.). Indeed, the optimal point for this transition is precisely the point at which the attacker "breaks even." Before this point it is preferable to use CAPTCHAs to minimize the cost burden to the site owner and the potential impact on legitimate users. While we do not believe that such economic models have been carefully developed by site owners, we see evidence that precisely this kind of tradeoff is being made. For example, a number of popular sites such as Google are now making aggressive use of secondary mechanisms to screen account sign-ups (e.g., SMS challenges), but *only* after a CAPTCHA is passed and some usage threshold is triggered (e.g., multiple sign-ups from the same IP address).[10]

In summary, we have argued that CAPTCHAs, while traditionally viewed as a *technological* impediment to an attacker, should more properly be regarded as an *economic* one, as witnessed by a robust and mature CAPTCHA-solving industry which bypasses the underly-

[9] These numbers should be taken with a grain of salt, both because the cited study is but a single data point, and because they studied SMTP-based spam, which generally has lower deliverability than Webmail-based spam. Anecdotally, the retail cost of Webmail-based delivery can be over 100 times more than via SMTP from raw bots.

[10] Anecdotally, this strategy appears effective for now and Gmail accounts on the underground market have gone from a typical asking price of $8/1,000, to being hard to come by at any price. We will not be surprised, however, if this mechanism leads to the monetization of smartphone botnets, or mobots [10], in response.

ing technological issue completely. Viewed in this light, CAPTCHAs are a low-impact mechanism that adds friction to the attacker's business model and thus minimizes the cost and legitimate user impact of heavier-weight secondary defenses. CAPTCHAs continue to serve this function, but as with most such defensive mechanisms, they simply work less efficiently over time.

## Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Rachna Dhamija, for their feedback as well as Luis von Ahn for his input and discussion early on in the project. We also thank Jonathan Wilkins for granting us access to reCaptchaOCR, Jon Howell and Jeremy Elson for discussions about the Asirra CAPTCHA, and the volunteers who assisted in manual identification of targeted CAPTCHAs. We are particularly indebted to MR. E for his generosity and time in answering our questions and sharing his insights about the technical and business aspects of operating a CAPTCHA-solving service. Finally we would also like to thank Anastasia Levchenko and Ilya Kolupaev for their assistance. This work was supported in part by National Science Foundation grants NSF-0433668 and NSF-0831138, by the Office of Naval Research MURI grant N000140911081, and by generous research, operational and in-kind support from Yahoo, Microsoft, HP, Google, and the UCSD Center for Networked Systems (CNS). McCoy was supported by a CCC-CRA-NSF Computing Innovation Fellowship.

## References

[1] BBC news PC stripper helps spam to spread. http://news.bbc.co.uk/2/hi/technology/7067962.stm.

[2] Ticketmaster, LLC v. RMG Technologies, Inc., et al 507 F.Supp.2d 1096 (C.D. Ca., October 16, 2007).

[3] E. Bursztein, S. Bethard, J. C. Mitchell, D. Jurafsky, and C. Fabry. How good are humans at solving CAPTCHAs? a large scale evaluation. In *IEEE S&P '10*, 2010.

[4] M. Chew and D. Tygar. Image recognition CAPTCHAs. In *Information Security, 7th International Conference, ISC 2004*, pages 268–279. Springer, 2004.

[5] D. Danchev. Inside India's CAPTCHA solving economy. http://blogs.zdnet.com/security/?p=1835, 2008.

[6] D. Danchev. Report: Google's reCAPTCHA flawed. http://blogs.zdnet.com/security/?p=5123, 2009.

[7] R. Datta, J. Li, and J. Z. Wang. Exploiting the Human-Machine Gap in Image Recognition for Designing CAPTCHAs. *IEEE Transactions on Information Forensics and Security*, 4(3):504–518, 2009.

[8] M. Egele, L. Bilge, E. Kirda, and C. Kruegel. CAPTCHA Smuggling: Hijacking Web Browsing Sessions to Create CAPTCHA Farms. In *The 25th Symposium On Applied Computing (SAC)*, pages 1865–1870. ACM, March 2010.

[9] J. Elson, J. R. Douceur, J. Howell, and J. Saul. Asirra: a CAPTCHA that exploits interest-aligned manual image categorization. In *CCS '07*, pages 366–374, New York, NY, USA, 2007. ACM.

[10] C. Fleizach, M. Liljenstam, P. Johansson, G. M. Voelker, and A. Méhes. Can You Infect Me Now? Malware Propagation in Mobile Phone Networks. In *Proceedings of the ACM Workshop on Recurring Malcode (WORM)*, Washington D.C., Nov. 2007.

[11] A. Hindle, M. W. Godfrey, and R. C. Holt. Reverse Engineering CAPTCHAs. In *Proc. of the 15th Working Conference on Reverse Engineering*, pages 59–68, 2008.

[12] L. Jassin-O'Rourke Group. Global Apparel Manufacturing Labor Cost Analysis 2008. http://www.tammonline.com/files/GlobalApparelLaborCostSummary2008.pdf, 2008.

[13] R. F. Jonell Baltazar, Joey Costoya. The heart of KOOBFACE: C&C and social network propagation. http://us.trendmicro.com/us/trendwatch/research-and-analysis/white-papers-and-articles/, October 2009.

[14] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *CCS '08*, pages 3–14, New York, NY, USA, 2008. ACM.

[15] The Klingon language institute. http://www.kli.org, Accessed February 2010.

[16] G. Mori and J. Malik. Recognizing objects in adversarial clutter: Breaking a visual CAPTCHA. In *CVPR*, volume 1, pages 134–141, 2003.

[17] G. Moy, N. Jones, C. Harkless, and R. Potter. Distortion estimation techniques in solving visual CAPTCHAs. pages II: 23–28, 2004.

[18] PWNTcha. Pretend We're Not a Turing computer but a human antagonist. http://caca.zoy.org/wiki/PWNtcha.

[19] G. Sauer, H. Hochheiser, J. Feng, and J. Lazar. Towards a universally usable CAPTCHA. In *SOUPS '08*, 2008.

[20] Symantec. A captcha-solving service. http://www.symantec.com/connect/blogs/captcha-solving-service.

[21] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. Captcha: Using hard AI problems for security. In *Advances in Cryptology - EUROCRYPT*, 2003.

[22] S.-Y. Wang, H. S. Baird, and J. L. Bentley. CAPTCHA challenge tradeoffs: Familiarity of strings versus degradation of images. In *ICPR '06*, 2006.

[23] J. Wilkins. Strong captcha guidelines v1.2. http://bitland.net/captcha.pdf.

[24] Xrumer. http://www.botmasternet.com/.

[25] J. Yan and A. S. El Ahmad. A low-cost attack on a Microsoft CAPTCHA. In *CCS '08*, pages 543–554, New York, NY, USA, 2008. ACM.

[26] J. Yan and A. S. El Ahmad. Usability of CAPTCHAs or usability issues in CAPTCHA design. In *SOUPS '08*, pages 44–52, New York, NY, USA, 2008. ACM.



# Chipping Away at Censorship Firewalls with User-Generated Content

*Sam Burnett, Nick Feamster, and Santosh Vempala*
*School of Computer Science, Georgia Tech*
{*sburnett, feamster, vempala*}*@cc.gatech.edu*

## Abstract

Oppressive regimes and even democratic governments restrict Internet access. Existing anti-censorship systems often require users to connect through proxies, but these systems are relatively easy for a censor to discover and block. This paper offers a possible next step in the censorship arms race: rather than relying on a single system or set of proxies to circumvent censorship firewalls, we explore whether the vast deployment of sites that host user-generated content can breach these firewalls. To explore this possibility, we have developed Collage, which allows users to exchange messages through hidden channels in sites that host user-generated content. Collage has two components: a message vector layer for embedding content in cover traffic; and a rendezvous mechanism to allow parties to publish and retrieve messages in the cover traffic. Collage uses user-generated content (*e.g.*, photo-sharing sites) as "drop sites" for hidden messages. To send a message, a user embeds it into cover traffic and posts the content on some site, where receivers retrieve this content using a sequence of tasks. Collage makes it difficult for a censor to monitor or block these messages by exploiting the sheer number of sites where users can exchange messages and the variety of ways that a message can be hidden. Our evaluation of Collage shows that the performance overhead is acceptable for sending small messages (*e.g.*, Web articles, email). We show how Collage can be used to build two applications: a direct messaging application, and a Web content delivery system.

## 1 Introduction

Network communication is subject to censorship and surveillance in many countries. An increasing number of countries and organizations are blocking access to parts of the Internet. The Open Net Initiative reports that 59 countries perform some degree of filtering [36]. For example, Pakistan recently blocked YouTube [47]. Content deemed offensive by the government has been blocked in Turkey [48]. The Chinese government regularly blocks activist websites [37], even as China has become the country with the most Internet users [19]; more recently, China has filtered popular content sites such as Facebook and Twitter, and even require their users to register to visit certain sites [43]. Even democratic countries such as the United Kingdom and Australia have recently garnered attention with controversial filtering practices [35, 54, 55]; South Korea's president recently considered monitoring Web traffic for political opposition [31].

Although existing anti-censorship systems—notably, onion routing systems such as Tor [18]—have allowed citizens some access to censored information, these systems require users outside the censored regime to set up infrastructure: typically, they must establish and maintain proxies of some kind. The requirement for running fixed infrastructure outside the firewall imposes two limitations: (1) a censor can discover and block the infrastructure; (2) benevolent users outside the firewall must install and maintain it. As a result, these systems are somewhat easy for censors to monitor and block. For example, Tor has recently been blocked in China [45]. Although these systems may continue to enjoy widespread use, this recent turn of events does beg the question of whether there are fundamentally new approaches to advancing this arms race: specifically, we explore whether it is possible to circumvent censorship firewalls with infrastructure that is more pervasive, and that does not require individual users or organizations to maintain it.

We begin with a simple observation: countless sites allow users to upload content to sites that they do not maintain or own through a variety of media, ranging from photos to blog comments to videos. Leveraging the large number of sites that allow users to upload their own content potentially yields many small cracks in censorship firewalls, because there are many different types of me-

dia that users can upload, and many different sites where they can upload it. The sheer number of sites that users could use to exchange messages, and the many different ways they could hide content, makes it difficult for a censor to successfully monitor and block all of them.

In this paper, we design a system to circumvent censorship firewalls using different types of user-generated content as cover traffic. We present Collage, a method for building message channels through censorship firewalls using user-generated content as the cover medium. Collage uses existing sites to host user-generated content that serves as the cover for hidden messages (*e.g.*, photo-sharing, microblogging, and video-sharing sites). Hiding messages in photos, text, and video across a wide range of host sites makes it more difficult for censors to block all possible sources of censored content. Second, because the messages are hidden in other seemingly innocuous messages, Collage provides its users some level of deniability that they do not have in using existing systems (*e.g.*, accessing a Tor relay node immediately implicates the user that contacted the relay). We can achieve these goals with minimal out-of-band communication.

Collage is not the first system to suggest using covert channels: much previous work has explored how to build a covert channel that uses images, text, or some other media as cover traffic, sometimes in combination with mix networks or proxies [3, 8, 17, 18, 21, 38, 41]. Other work has also explored how these schemes might be broken [27], and others hold the view that message hiding or "steganography" can never be fully secure. Collage's new contribution, then, is to design covert channels based on user-generated content and imperfect message-hiding techniques in a way that circumvents censorship firewalls that is robust enough to allow users to freely exchange messages, even in the face of an adversary that may be looking for such suspicious cover traffic.

The first challenge in designing Collage is to develop an appropriate *message vector* for embedding messages in user-generated content. Our goal for developing a message vector is to find user-generated traffic (*e.g.*, photos, blog comments) that can act as a cover medium, is widespread enough to make it difficult for censors to completely block and remove, yet is common enough to provide users some level of deniability when they download the cover traffic. In this paper, we build message vectors using the user-generated photo-sharing site, Flickr [24], and the microblogging service, Twitter [49], although our system in no way depends on these particular services. We acknowledge that some or all of these two specific sites may ultimately be blocked in certain countries; indeed, we witnessed that parts of Flickr were already blocked in China when accessed via a Chinese proxy in January 2010. A main strength of Collage's design is that blocking a specific site or set of sites will not

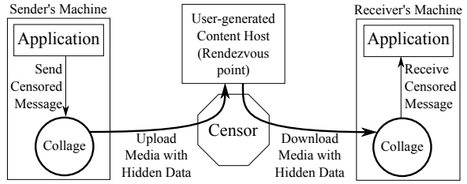



Figure 1: Collage's interaction with the network. See Figure 2 for more detail.

fully stem the flow of information through the firewall, since users can use so many sites to post user-generated content. We have chosen Flickr and Twitter as a proof of concept, but Collage users can easily use domestic equivalents of these sites to communicate using Collage.

Given that there are necessarily many places where one user might hide a message for another, the second challenge is to agree on *rendezvous* sites where a sender can leave a message for a receiver to retrieve. We aim to build this *message layer* in a way that the client's traffic looks innocuous, while still preventing the client from having to retrieve an unreasonable amount of unwanted content simply to recover the censored content. The basic idea behind rendezvous is to embed message segments in enough cover material so that it is difficult for the censor to block all segments, even if it joins the system as a user; and users can retrieve censored messages without introducing significant deviations in their traffic patterns. In Collage, senders and receivers agree on a common set of network locations where any given content should be hidden; these agreements are established and communicated as "tasks" that a user must perform to retrieve the content (*e.g.*, fetching a particular URL, searching for content with a particular keyword). Figure 1 summarizes this process. Users send a message with three steps: (1) divide a message into many erasure-encoded "blocks" that correspond to a task, (2) embed these blocks into user-generated content (*e.g.*, images), and (3) publish this content at user-generated content sites, which serve as rendezvous points between senders and receivers. Receivers then retrieve a subset of these blocks to recover the original message by performing one of these tasks.

This paper presents the following contributions.

- We present the design and implementation of Collage, a censorship-resistant message channel built using user-generated content as the cover medium. An implementation of the Collage message channel is publicly available [13].
- We evaluate the performance and security of Collage. Collage does impose some overhead, but the overhead is acceptable for small messages (*e.g.*, articles, emails, short messages), and Collage's overhead can also be reduced at the cost of making the system less robust to blocking.
- We present Collage's general message-layer abstraction and show how this layer can serve as the foundation for two different applications: Web publishing and direct messaging (*e.g.*, email). We describe and evaluate these two applications.

The rest of this paper proceeds as follows. Section 2 presents related work. In Section 3, we describe the design goals for Collage and the capabilities of the censor. Section 4 presents the design and implementation of Collage. Section 5 evaluates the performance of Collage's messaging layer and applications. Section 6 describes the design and implementation of two applications that are built on top of this messaging layer. Section 7 discusses some limitations of Collage's design and how Collage might be extended to cope with increasingly sophisticated censors. Section 8 concludes.

## 2 Background and Related Work

We survey other systems that provide anonymous, confidential, or censorship-resistant communication. We note that most of these systems require setting up a dedicated infrastructure of some sort, typically based on proxies. Collage departs significantly from this approach, since it leverages existing infrastructure. At the end of this section, we discuss some of the challenges in building covert communications channels using existing techniques, which have also been noted in previous work [15].

**Anonymization proxies.** Conventional anti-censorship systems have typically consisted of simple Web proxies. For example, Anonymizer [3] is a proxy-based system that allows users to connect to an anonymizing proxy that sits outside a censoring firewall; user traffic to and from the proxy is encrypted. These types of systems provide confidentiality, but typically do not satisfy any of the other design goals: for example, the existence of any encrypted traffic might be reason for suspicion (thus violating deniability), and a censor that controls a censoring firewall can easily block or disrupt communication once the proxy is discovered (thus violating resilience). A censor might also be able to use techniques such as SSL fingerprinting or timing attacks to link senders and receivers, even if the underlying traffic is encrypted [29]. Infranet attempts to create deniability for clients by embedding censored HTTP requests and content in HTTP traffic that is statistically indistinguishable from "innocuous" HTTP traffic [21]. Infranet improves deniability, but it still depends on cooperating proxies outside the firewall that might be discovered and blocked by censors. Collage improves availability by leveraging the large number of user-generated content sites, as opposed to a relatively smaller number of proxies.

One of the difficult problems with anti-censorship proxies is that a censor could also discover these proxies and block access to them. Feamster *et al.* proposed a proxy-discovery method based on frequency hopping [22]. Kaleidoscope is a peer-to-peer overlay network to provide users robust, highly available access to these proxies [42]. This system is complementary to Collage, as it focuses more on achieving availability, at the expense of deniability. Collage focuses more on providing users deniability and preventing the censor from locating all hosts from where censored content might be retrieved.

**Anonymous publishing and messaging systems.** CovertFS [5] is a file system that hides data in photos using steganography. Although the work briefly mentions challenges in deniability and availability, it is easily defeated by many of the attacks discussed in Section 7. Furthermore, CovertFS could in fact be implemented using Collage, thereby providing the design and security benefits described in this paper.

Other existing systems allow publishers and clients to exchange content using either peer-to-peer networks (Freenet [12]) or using a storage system that makes it difficult for an attacker to censor content without also removing legitimate content from the system (Tangler [53]). Freenet provides anonymity and unlinkability, but does not provide deniability for users of the system, nor does it provide any inherent mechanisms for resilience: an attacker can observe the messages being exchanged and disrupt them in transit. Tangler's concept of document entanglement could be applied to Collage to prevent the censor from discovering which images contain embedded information.

**Anonymizing mix networks.** Mix networks (*e.g.*, Tor [18], Tarzan [25], Mixminion [17]) offer a network of machines through which users can send traffic if they wish to communicate anonymously with one another. Danezis and Dias present a comprehensive survey of these networks [16]. These systems also attempt to provide unlinkability; however, previous work has shown that, depending on its location, a censor or observer might be able to link sender and receiver [4, 6, 23, 33, 39, 40]. These systems also do not provide deniability for users, and typically focus on anonymous point-to-point communication. In contrast, Collage provides a deniable means for asynchronous point-to-point communication. Finally, mix networks like Tor traditionally use a pub-

lic relay list which is easily blocked, although work has been done to try to rectify this [44, 45].

**Message hiding and embedding techniques.** Collage relies on techniques that can embed content into cover traffic. The current implementation of Collage uses an image steganography tool called `outguess` [38] for hiding content in images and a text steganography tool called SNOW [41] for embedding content in text. We recognize that steganography techniques offer no formal security guarantees; in fact, these schemes can and have been subject to various attacks (*e.g.*, [27]). Danezis has also noted the difficulty in building covert channels with steganography alone [15]: not only can the algorithms be broken, but also they do not hide the identities of the communicating parties. Thus, these functions must be used as components in a larger system, not as standalone "solutions". Collage relies on the embedding functions of these respective algorithms, but its security properties do not hinge solely on the security properties of any single information hiding technique; in fact, Collage could have used watermarking techniques instead, but we chose these particular embedding techniques for our proof of concept because they had readily available, working implementations. One of the challenges that Collage's design addresses is how to use imperfect message hiding techniques to build a message channel that is both available and offers some amount of deniability for users.

# 3 Problem Overview

We now discuss our model for the censor's capabilities and our goals for circumventing a censor who has these capabilities. It is difficult, if not impossible, to fully determine the censor's current or potential capabilities; as a result, Collage cannot provide formal guarantees regarding success or deniability. Instead, we present a model for the censor that we believe is more advanced than current capabilities and, hence, where Collage is *likely* to succeed. Nevertheless, censorship is an arms race, so as the censor's capabilities evolve, attacks against censorship firewalls will also need to evolve in response. In Section 7, we discuss how Collage's could be extended to deal with these more advanced capabilities as the censor becomes more sophisticated.

We note that although we focus on censors, Collage also depends on content hosts to store media containing censored content. Content hosts currently do not appear to be averse to this usage (*e.g.*, to the best of our knowledge, Collage does not violate the Terms of Service for either Flickr or Twitter), although if Collage were to become very popular this attitude would likely change. Although we would prefer content hosts to willingly serve Collage content (*e.g.*, to help users in censored regimes), Collage can use many content hosts to prevent any single host from compromising the entire system.

## 3.1 The Censor

We assume that the censor wishes to allow *some* Internet access to clients, but can monitor, analyze, block, and alter subsets of this traffic. We believe this assumption is reasonable: if the censor builds an entirely separate network that is partitioned from the Internet, there is little we can do. Beyond this basic assumption, there is a wide range of capabilities we can assume. Perhaps the most difficult aspect of modeling the censor is figuring out how much effort it will devote to capturing, storing, and analyzing network traffic. Our model assumes that the censor can deploy monitors at multiple network egress points and observe all traffic as it passes (including both content and headers). We consider two types of capabilities: targeting and disruption.

**Targeting.** A censor might *target* a particular user behind the firewall by focusing on that user's traffic patterns; it might also target a particular suspected content host site by monitoring changes in access patterns to that site (or content on that site). In most networks, a censor can monitor all traffic that passes between its clients and the Internet. Specifically, we assume the censor can eavesdrop any network traffic between clients on its network and the Internet. A censor's motive in passively monitoring traffic would most likely be either to determine that a client was using Collage or to identify sites that are hosting content. To do so, the censor could monitor traffic *aggregates* (*i.e.*, traffic flow statistics, like NetFlow [34]) to determine changes in overall traffic patterns (*e.g.*, to determine if some website or content has suddenly become more popular). The censor can also observe traffic streams from *individual* users to determine if a particular user's clickstream is suspicious, or otherwise deviates from what a real user would do. These capabilities lead to two important requirements for preserving deniability: traffic patterns generated by Collage should not skew overall distributions of traffic, and the traffic patterns generated by an individual Collage user must resemble the traffic generated by innocuous individuals.

To target users or sites, a censor might also use Collage as a sender or receiver. This assumption makes some design goals more challenging: a censor could, for example, inject bogus content into the system in an attempt to compromise message availability. It could also join Collage as a client to discover the locations of censored content, so that it could either block content outright (thus attacking availability) or monitor users who download similar sets of content (thus attacking deniability). We also assume that the censor could act as a content publisher. Finally, we assume that a censor might be able to coerce a content host to shut down its site (an aggressive variant of actively blocking requests to a site).

**Disruption.** A censor might attempt to *disrupt* communications by actively mangling traffic. We assume the censor would not mangle uncensored content in any way that a user would notice. A censor could, however, inject additional traffic in an attempt to confuse Collage's process for encoding or decoding censored content. We assume that it could also block traffic at granularities ranging from an entire site to content on specific sites.

**The costs of censorship.** In accordance with Bellovin's recent observations [7], we assume that the censor's capabilities, although technically limitless, will ultimately be constrained by cost and effort. In particular, we assume that the censor will *not* store traffic indefinitely, and we assume that the censor's will or capability to analyze traffic prevents it from observing more complex statistical distributions on traffic (*e.g.*, we assume that it cannot perform analysis based on joint distributions between arbitrary pairs or groups of users). We also assume that the censor's computational capabilities are limited: for example, performing deep packet inspection on every packet that traverses the network or running statistical analysis against all traffic may be difficult or infeasible, as would performing sophisticated timing attacks (*e.g.*, examining inter-packet or inter-request timing for each client may be computationally infeasible or at least prohibitively inconvenient). As the censorship arms race continues, the censor may develop such capabilities.

## 3.2 Circumventing the Censor

Our goal is to allow users to send and receive messages across a censorship firewall that would otherwise be blocked; we want to enable users to communicate across the firewall by exchanging articles and short messages (*e.g.*, email messages and other short messages). In some cases, the sender may be behind the firewall (*e.g.*, a user who wants to publish an article from within a censored regime). In other cases, the receiver might be behind the firewall (*e.g.*, a user who wants to browse a censored website).

We aim to understand Collage's performance in real applications and demonstrate that it is "good enough" to be used in situations where users have no other means for circumventing the firewall. We therefore accept that our approach may impose substantial overhead, and we do not aim for Collage's performance to be comparable to that of conventional networked communication. Ultimately, we strive for a system that is effective and easy to use for a variety of networked applications. To this end, Collage offers a messaging library that can support these applications; Section 6 describes two example applications.

Collage's main performance requirement is that the overhead should be small enough to allow content to be stored on sites that host user-generated content and to allow users to retrieve the hidden content in a reasonable amount of time (to ensure that the system is usable), and with a modest amount of traffic overhead (since some users may be on connections with limited bandwidth). In Section 5, we evaluate Collage's storage requirements on content hosting sites, the traffic overhead of each message (as well as the tradeoff between this overhead and robustness and deniability), and the overall transfer time for messages.

In addition to performance requirements, we want Collage to be robust in the face of the censor that we have outlined in Section 3.1. We can characterize this robustness in terms of two more general requirements. The first requirement is *availability*, which says that clients should be able to communicate in the face of a censor that is willing to restrict access to various content and services. Most existing censorship circumvention systems do not prevent a censor from blocking access to the system altogether. Indeed, regimes such as China have blocked or hijacked applications ranging from websites [43] to peer-to-peer systems [46] to Tor itself [45]. We aim to satisfy availability in the face of the censor's targeting capabilities that we described in Section 3.1.

Second, Collage should offer users of the system some level of *deniability*; although this design goal is hard to quantify or formalize, informally, deniability says that the censor cannot discover the users of the censorship system. It is important for two reasons. First, if the censor can identify the traffic associated with an anti-censorship system, it can discover and either block or hijack that traffic. As mentioned above, a censor observing encrypted traffic may still be able to detect and block systems such as Tor [18]. Second, and perhaps more importantly, if the censor can identify specific users of a system, it can coerce those users in various ways. Past events have suggested that censors are able and willing to both discover and block traffic or sites associated with these systems *and* to directly target and punish users who attempt to defeat censorship. In particular, China requires users to register with ISPs before purchasing Internet access at either home or work, to help facilitate tracking individual users [10]. Freedom House reports that in six of fifteen countries they assessed, a blogger or online journalist was sentenced to prison for attempting to circumvent censorship laws—prosecutions have occurred in Tunisia, Iran, Egypt, Malaysia, and India [26]—and cites a recent event of a Chinese blogger who was recently attacked [11]. As these regimes have indicated

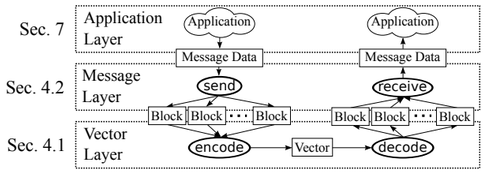



Figure 2: Collage's layered design model. Operations are in ovals; intermediate data forms are in rectangles.

their willingness and ability to monitor and coerce individual users, we believe that attempting to achieve some level of deniability is important for any anti-censorship system.

By design, a user cannot disprove claims that he engages in deniable communication, thus making it easier for governments and organizations to implicate arbitrary users. We accept this as a potential downside of deniable communications, but point out that organizations can already implicate users with little evidence (*e.g.*, [2]).

## 4 Collage Design and Implementation

Collage's design has three layers and roughly mimics the layered design of the network protocol stack itself. Figure 2 shows these three layers: the vector, message, and application layers. The *vector layer* provides storage for short data chunks (Section 4.1), and the *message layer* specifies a protocol for using the vector layer to send and receive messages (Section 4.2). A variety of applications can be constructed on top of the message layer. We now describe the vector and message layers in detail, deferring discussion of specific applications to Section 6. After describing each of these layers, we discuss *rendezvous*, the process by which senders and receivers find each other to send messages using the message layer (Section 4.3). Finally, we discuss our implementation and initial deployment (Section 4.4).

### 4.1 Vector Layer

The vector layer provides a substrate for storing short data chunks. Effectively, this layer defines the "cover media" that should be used for embedding a message. For example, if a small message is hidden in the high frequency of a video then the *vector* would be, for example, a YouTube video. This layer hides the details of this choice from higher layers and exposes three operations: encode, decode, and isEncoded. These operations encode data into a vector, decode data from an encoded vector, and check for the presence of encoded data given a secret key, respectively.

Collage imposes requirements on the choice of vector. First, each vector must have some capacity to hold encoded data. Second, the population of vectors must be large so that many vectors can carry many messages. Third, to satisfy both availability and deniability, it must be relatively easy for users to deniably send and receive vectors containing encoded chunks. Fourth, to satisfy availability, it must be expensive for the censor to disrupt chunks encoded in a vector. Any vector layer with these properties will work with Collage's design, although the deniability of a particular application will also depend upon its choice of vector, as we discuss in Section 7.

The feasibility of the vector layer rests on a key observation: *data hidden in user-generated content serves as a good vector for many applications, since it is both populous and comes from a wide variety of sources* (i.e., *many users)*. Examples of such content include images published on Flickr [24] (as of June 2009, Flickr had about 3.6 billion images, with about 6 million new images per day [28]), tweets on Twitter [49] (Twitter had about half a million tweets per day [52], and Mashable projected about 18 million Twitter users by the end of 2009 [50]), and videos on YouTube [56], which had about $200,000$ new videos per day as of March 2008 [57].

For concreteness, we examine two classes of vector encoding algorithms. The first option is *steganography*, which attempts to hide data in a cover medium such that only intended recipients of the data (*e.g.*, those possessing a key) can detect its presence. Steganographic techniques can embed data in a variety of cover media, such as images, video, music, and text. Steganography makes it easy for legitimate Collage users to find vectors containing data and difficult for a censor to identify (and block) encoded vectors. Although the deniability that steganography can offer is appealing, key distribution is challenging, and almost all production steganography algorithms have been broken. Therefore, we cannot simply rely on the security properties of steganography.

Another option for embedding messages is *digital watermarking*, which is similar to steganography, except that instead of hiding data from the censor, watermarking makes it difficult to remove the data without destroying the cover material. Data embedded using watermarking is perhaps a better choice for the vector layer: although encoded messages are clearly visible, they are difficult to remove without destroying or blocking a large amount of legitimate content. If watermarked content is stored in a lot of popular user-generated content, Collage users can gain some level of deniability simply because all popular content contains some message chunks.

We have implemented two example vector layers. The first is image steganography applied to images hosted on Flickr [24]. The second is text steganography applied to user-generated text comments on websites such as blogs, YouTube [56], Facebook [20], and Twitter [49]. Despite possible and known limitations to these approaches (*e.g.*, [27]), both of these techniques have working implementations with running code [38, 41]. As watermarking and other data-hiding techniques continue to become more robust to attack, and as new techniques and implementations emerge, Collage's layered model can incorporate those mechanisms. The goal of this paper is *not* to design better data-hiding techniques, but rather to build a censorship-resistant message channel that leverages these techniques.

```
send(identifier, data)
1  Create a rateless erasure encoder for data.
2  for each suitable vector (e.g., image file)
3      do
4          Retrieve blocks from the erasure coder to
               meet the vector's encoding capacity.
5          Concatenate and encrypt these blocks using
               the identifier as the encryption key.
6          encode the ciphertext into the vector.
7          Publish the vector on a user-generated
               content host such that receivers
               can find it. See Section 4.3.
```

```
receive(identifier)
1  Create a rateless erasure decoder.
2  while the decoder cannot decode the message
3      do
4          Find and fetch a vector from a
               user-generated content host.
5          Check if the vector contains encoded
               data for this identifier.
6          if the vector is encoded with message data
7            then
8                 decode payload from the vector.
9                 Decrypt the payload.
10                Split the plaintext into blocks.
11                Provide each decrypted block to
                      the erasure decoder.
12 return decoded message from erasure decoder
```

Figure 3: The message layer's send and receive operations.

### 4.2 Message Layer

The message layer specifies a protocol for using the vector layer to send and receive arbitrarily long messages (*i.e.*, exceeding the capacity of a single vector). Observable behavior generated by the message layer should be deniable with respect to the normal behavior of the user or users at large.

Figure 3 shows the send and receive operations. send encodes message data in vectors and publishes them on content hosts, while receive finds encoded vectors on content hosts and decodes them to recover the original message. The sender associates a message identifier with each message, which should be unique for an application (*e.g.*, the hash of the message). Receivers use this identifier to locate the message. For encoding schemes that require a key (*e.g.*, [38]), we choose the key to be the message identifier.

To distribute message data among several vectors, the protocol uses rateless erasure coding [9, 32], which generates a near-infinite supply of short chunks from a source message such that any appropriately-sized subset of those chunks can reassemble the original message. For example, a rateless erasure coder could take a $80$ KB message and generate $1$ KB chunks such that any $100$-subset of those chunks recovers the original message. Step 1 of send initializes a rateless erasure encoder for generating chunks of the message; step 4 retrieves chunks from the encoder. Likewise, step 1 of receive creates a rateless erasure decoder, step 11 provides retrieved chunks to the decoder, and step 12 recovers the message.

Most of the remaining send operations are straightforward, involving encryption and concatenation (step 5), and operation of the vector layer's encode function (step 6). Likewise, receive operates the vector layer's decode function (step 8), decrypts and splits the payload (steps 9 and 10). The only more complex operations are step 7 of send and step 4 of receive, which publish and retrieve content from user-generated content hosts. These steps must ensure (1) that senders and receivers agree on locations of vectors and (2) that publishing and retrieving vectors is done in a deniable manner. We now describe how to meet these two requirements.

### 4.3 Rendezvous: Matching Senders to Receivers

Vectors containing message data are stored to and retrieved from user-generated content hosts; to exchange messages, senders and receivers must first *rendezvous*. To do so, senders and receivers perform sequences of *tasks*, which are time-dependent sequences of actions. An example of a sender task is the sequence of HTTP requests (*i.e.*, actions) and fetch times corresponding to "Upload photos tagged with 'flowers' to Flickr"; a corresponding receiver task is "Search Flickr for photos tagged with 'flowers' and download the first 50 images." This scheme poses many challenges: (1) to achieve deniability, all tasks must resemble observable actions completed by innocuous entities not using Collage (*e.g.*, browsing the Web), (2) senders must identify vectors suitable for each task, and (3) senders and receivers must

agree on which tasks to use for each message. This section addresses these challenges.

**Identifying suitable vectors.** Task deniability depends on properly selecting vectors for each task. For example, for the receiver task "search for photos with keyword *flowers*," the corresponding sender task ("publish a photo with keyword *flowers*") must be used with photos of flowers; otherwise, the censor could easily identify vectors containing Collage content as those vectors that do not match their keywords. To achieve this, the sender picks vectors with attributes (*e.g.*, associated keywords) that match the expected content of the vector.

**Agreeing on tasks for a message.** Each user maintains a list of deniable tasks for common behaviors involving vectors (Section 4.1) and uses this list to construct a *task database*. The database is simply a table of pairs $(T_s, T_r)$, where $T_s$ is a sender task and $T_r$ is a receiver task. Senders and receivers construct pairs such that $T_s$ publishes vectors in locations visited by $T_r$. For example, if $T_r$ performs an image search for photos with keyword "flowers" then $T_s$ would publish only photos with that keyword (and actually depicting flowers). Given this database, the sender and receiver map each message identifier to one or more task pairs and execute $T_s$ and $T_r$, respectively.

The sender and receiver must agree on the mapping of identifiers to database entries; otherwise, the receiver will be unable to find vectors published by the sender. If the sender's and receiver's databases are identical, then the sender and receiver simply use the message identifier as an index into the task database. Unfortunately, the database may change over time, for a variety of reasons: tasks become obsolete (*e.g.*, Flickr changes its page structure) and new tasks are added (*e.g.*, it may be advantageous to add a task for a new search keyword during a current event, such as an election). Each time the database changes, other users need to be made aware of these changes. To this end, Collage provides two operations on the task database: add and remove. When a user receives an advertisement for a new task or a withdrawal of an existing task he uses these operations to update his copy of the task database.

Learning task advertisements and withdrawals is application specific. For some applications, a central authority sends updates using Collage's own message layer, while in others updates are sent offline (*i.e.*, separate from Collage). We discuss these options in Section 6. One feature is common to all applications: delays in propagation of database updates will cause different users to have slightly different versions of the task database, necessitating a mapping for identifiers to tasks that is robust to slight changes to the database.

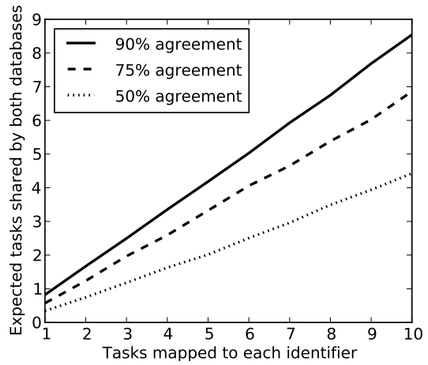



Figure 4: The expected number of common tasks when mapping the same message identifier to a task subset, between two task databases that agree on varying percentages of tasks.

To reconcile database disagreements, our algorithm for mapping message identifiers to task pairs uses *consistent hash functions* [30], which guarantee that small changes to the space of output values have minimal impact on the function mapping. We initialize the task database by choosing a pseudorandom hash function $h$ (*e.g.*, SHA-1) and precomputing $h(t)$ for each task $t$. The algorithm for mapping an identifier $M$ to a $m$-subset of the database is simple: compute $h(M)$ and take the $m$ entries from the task database with precomputed hash values closest to $h(M)$; these task pairs are the mapping for $M$.

Using consistent hashing to map identifiers to task pairs provides an important property: updating the database results in only small changes to the mappings for existing identifiers. Figure 4 shows the expected number of tasks reachable after removing a percentage of the task database and replacing it with new tasks. As expected, increasing the number of tasks mapped for each identifier decreases churn. Additionally, even if half of the database is replaced, the sender and receiver can agree on at least one task when three or more tasks are mapped to each identifier. In practice, we expect the difference between two task databases to be around 10%, so three tasks to each identifier is sufficient. Thus, two parties with slightly different versions of the task database can still communicate messages: although some tasks performed by the receiver (*i.e.*, mapped using his copy of the database) will not yield content, most tasks will.

**Choosing deniable tasks.** Tasks should mimic the normal behavior of users, so that a user who is performing these tasks is unlikely to be pinpointed as a Collage user (which, in and of itself, could be incriminating). We design task sequences to "match" those of normal visitors to user-generated content sites. Tasks for different content hosts have different deniability criteria. For example, the task of looking at photos corresponding to a popular tag or tag pair offers some level of deniability, because an innocuous user might be looking at popular images anyway. The challenge, of course, is finding sets of tasks that are deniable, yet focused enough to allow a user to retrieve content in a reasonable amount of time. We discuss the issue of deniability further in Section 7.

## 4.4 Implementation

Collage requires minimal modification to existing infrastructure, so it is small and self-contained, yet modular enough to support many possible applications; this should facilitate adoption. We have released a version of Collage [13].

We have implemented Collage as a 650-line Python library, which handles the logic of the message layer, including the task database, vector encoding and decoding, and the erasure coding algorithm. To execute tasks, the library uses Selenium [1], a popular web browser automation tool; Selenium visits web pages, fills out forms, clicks buttons and downloads vectors. Executing tasks using a real web browser frees us from implementing an HTTP client that produces realistic Web traffic (*e.g.*, by loading external images and scripts, storing cookies, and executing asynchronous JavaScript requests).

We represent tasks as Python functions that perform the requisite task. Table 1 shows four examples. Each application supplies definitions of operations used by the tasks (*e.g.*, `FindPhotosOfFlickrUser`). The task database is a list of tasks, sorted by their MD5 hash; to map an identifier to a set of tasks, the database finds the tasks with hashes closest to the hash of the message identifier. After mapping, receivers simply execute these tasks and decode the resulting vectors. Senders face a more difficult task: they must supply the task with a vector suitable for that task. For instance, the task "publish a photo tagged with 'flowers'" must be supplied with a photo of flowers. We delegate the task of finding vectors meeting specific requirements to a *vector provider*. The exact details differ between applications; one of our applications searches a directory of annotated photos, while another prompts the user to type a phrase containing certain words (*e.g.*, "Olympics").

# 5 Performance Evaluation

This section evaluates Collage according to the three performance metrics introduced in Section 3: storage overhead on content hosts, network traffic, and transfer time. We characterize Collage's performance by measuring its behavior in response to a variety of parameters. Recall that Collage (1) processes a message through an erasure coder, (2) encodes blocks inside vectors, (3) executes tasks to distribute the message vectors to content hosts, (4) retrieves some of these vectors from content hosts, and (5) decodes the message on the receiving side. Each stage can affect performance. In this section, we evaluate how each of these factors affects the performance of the message layer; Section 6 presents additional performance results for Collage applications using real content hosts.

- **Erasure coding** can recover an $n$-block message from $(1+\frac{\epsilon}{2})n$ of its coded message blocks. Collage uses $\epsilon = 0.01$, as recommended by [32], yielding an expected $0.5\%$ increase in storage, traffic, and transfer time of a message.
- **Vector encoding** stores erasure coded blocks inside vectors. Production steganography tools achieve encoding rates of between $0.01$ and $0.05$, translating to between 20 and 100 factor increases in storage, traffic, and transfer time [38]. Watermarking algorithms are less efficient; we hope that innovations in information hiding can reduce this overhead.
- **Sender and receiver tasks** publish and retrieve vectors from content hosts. Tasks do not affect the storage requirement on content hosts, but each task can impose additional traffic and time. For example, a task that downloads images by searching for them on Flickr can incur hundreds of kilobytes of traffic before finding encoded vectors. Depending on network connectivity, this step could take anywhere from a few seconds to a few minutes and can represent an overhead of several hundred percent, depending on the size of each vector.
- **The number of executed tasks** differs between senders and receivers. The receiver performs as many tasks as necessary until it is able to decode the message; this number depends on the size of the message, the number of vectors published by the sender, disagreements between sender and receiver task databases, the dynamics of the content host (*e.g.*, a surge of Flickr uploads could "bury" Collage encoded vectors), and the number of tasks and vectors blocked by the censor. While testing Collage, we found that we needed to execute only one task for the majority of cases.

  The sender must perform as many tasks as necessary so that, given the many ways the receiver can fail to obtain vectors, the receiver will still be able to retrieve enough vectors to decode the message. In practice, this number is difficult to estimate and

| Content host | Sender task | Receiver task |
|---|---|---|
| Flickr | PublishAsUser(‘User’, Photo, MsgData) | FindPhotosOfFlickrUser(‘User’) |
| Twitter | PostTweet(‘Watching the Olympics’, MsgData) | SearchTwitter(‘Olympics’) |

Table 1: Examples of sender and receiver task snippets.

vectors are scarce, so the sender simply uploads as many vectors as possible.

We implemented a Collage application that publishes vectors on a simulated content host, allowing us to observe the effects of these parameters. Figure 5 shows the results of running several experiments across Collage's parameter space. The simulation sends and receives a 23 KB one-day news summary. The message is erasure coded with a block size of 8 bytes and encoded into several vectors randomly drawn from a pool for vectors with average size 200 KB. Changing the message size scales the metrics linearly, while increasing the block size only decreases erasure coding efficiency.

Figure 5a demonstrates the effect of vector encoding efficiency on required storage on content hosts. We used a fixed-size identifier-to-task mapping of ten tasks. We chose four *send rates*, which are multiples of the minimum number of tasks required to decode the message: the sender may elect to send more vectors if he believes some vectors may be unreachable by the receiver. For example, with a send rate of 10x, the receiver can still retrieve the message even if 90% of vectors are unavailable. Increasing the task mapping size may be necessary for large send rates, because sending more vectors requires executing more tasks. These results give us hope for the future of information hiding technology: current vector encoding schemes are around 5% efficient; according to Figure 5a, this a region where a significant reduction in storage is possible with only incremental improvements in encoding techniques (*i.e.*, the slope is steep).

Figure 5b predicts total sender and receiver traffic from task overhead traffic, assuming 1 MB of vector storage on the content host. As expected, blocking more vectors increases traffic, as the receiver must execute more tasks to receive the same message content. Increasing storage beyond 1 MB decreases receiver traffic, because more message vectors are available for the same blocking rate. An application executed on a real content host transfers around 1 MB of overhead traffic for a 23 KB message.

Finally, Figure 5c shows the overall transfer time for senders and receivers, given varying time overheads. These overheads are optional for both senders and receivers and impose delays between requests to evade timing analysis by the censor. For example, Collage could build a distribution of inter-request timings from the user's normal (*i.e.*, non-Collage) traffic and impose this timing distribution on Collage tasks. We simulated the total transfer time using three network connection speeds. The first (768 Kbps download and 384 Kbps upload) is a typical entry-level broadband package and would be experienced if both senders and receivers are typical users within the censored domain. The second (768/10000 Kbps) would be expected if the sender has a high-speed connection, perhaps operating as a dedicated publisher outside the censored domain; one of the applications in Section 6 follows this model. Finally, the 6000/1000 Kbps connection represents expected next-generation network connectivity in countries experiencing censorship. In all cases, reasonable delays are imposed upon transfers, given the expected use cases of Collage (*e.g.*, fetching daily news article). We confirmed this result: a 23 KB message stored on a real content host took under 5 minutes to receive over an unreliable broadband wireless link; sender time was less than 1 minute.

## 6 Building Applications with Collage

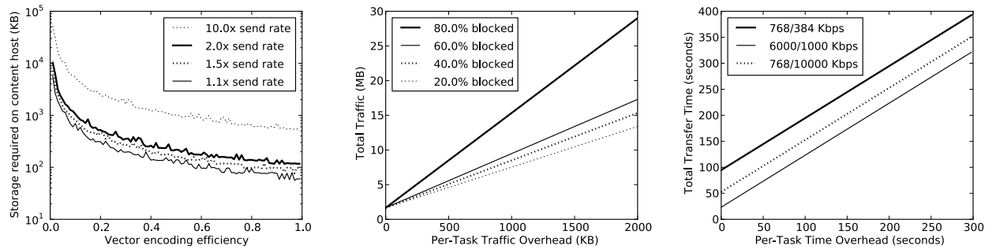
Developers can build a variety of applications using the Collage message channel. In this section, we outline requirements for using Collage and present two example applications.

### 6.1 Application Requirements

Even though application developers use Collage as a secure, deniable messaging primitive, they must still remain conscious of overall application security when using these primitives. Additionally, the entire vector layer and several parts of the message layer presented in Section 4 must be provided by the application. These components can each affect correctness, performance, and security of the entire application. In this section, we discuss each of these components. Table 2 summarizes the component choices.

**Vectors, tasks, and task databases.** Applications specify a class of vectors and a matching vector encoding algorithm (*e.g.*, Flickr photos with image steganography) based on their security and performance characteristics. For example, an application requiring strong content deniability for large messages could use a strong steganography algorithm to encode content inside of videos.






(a) Storage, for various sender redundancies (b) Traffic, for various vector block rates (c) Transfer time, for various network connectivity rates (download/upload)

Figure 5: Collage's performance metrics, as measured using a simulated content host.

| | Web Content Proxy (Sec. 6.2) | Covert Email (Sec. 6.3) | Other options |
|---|---|---|---|
| Vectors | Photos | Text | Videos, music |
| Vector encoding | Image steganography | Text steganography | Video steganography, digital watermarking |
| Vector sources | Users of content hosts | Covert Email users | Automatic generation, crawl the Web |
| Tasks | Upload/download Flickr photos | Post/receive Tweets | Other user-generated content host(s) |
| Database distribution | Send by publisher via proxy | Agreement by users | Prearranged algorithm, “sneakernet” |
| Identifier security | Distributed by publisher, groups | Group key | Existing key distribution infrastructure |

Table 2: Summary of application components.

Tasks are application-specific: uploading photos to Flickr is different from posting tweets on Twitter. Applications insert tasks into the task database, and the message layer executes these tasks when sending and receiving messages. The applications specify how many tasks are mapped to each identifier for database lookups. In Section 4.3, we showed that mapping each identifier to three tasks ensures that, on average, users can still communicate even with slightly out-of-date databases; applications can further boost availability by mapping more tasks to each identifier.

Finally, applications must distribute the task database. In some instances, a central authority can send the database to application users via Collage itself. In other cases, the database is communicated offline. The application's task database should be large enough to ensure diversity of tasks for messages published at any given time; if $n$ messages are published every day, then the database should have $cn$ tasks, where $c$ is at least the size of the task mapping. Often, tasks can be generated programmatically, to reduce network overhead. For example, our Web proxy (discussed next) generates tasks from a list of popular Flickr tags.

**Sources of vectors.** Applications must acquire vectors used to encode messages, either by requiring end-users to provide their own vectors (*e.g.*, from a personal photo collection), automatically generating them, or obtaining them from an external source (*e.g.*, a photo donation system).

**Identifier security.** Senders and receivers of a message must agree on a message identifier for that message. This process is analogous to key distribution. There is a general tradeoff between ease of message identifier distribution and security of the identifier: if users can easily learn identifiers, then more users will use the system, but it will also be easier for the censor to obtain the identifier; the inverse is also true. Developers must choose a distribution scheme that meets the intended use of their application. We discuss two approaches in the next two sections, although there are certainly other possibilities.

**Application distribution and bootstrapping.** Users ultimately need a secure one-time mechanism for obtaining the application, without using Collage. A variety of distribution mechanisms are possible: clients could receive software using spam or malware as a propagation vector, or via postal mail or person-to-person exchange. There will ultimately be many ways to distribute applications without the knowledge of the censor. Other systems face the same problem [21]. This requirement does not obviate Collage, since once the user has received the software, he or she can use it to exchange an arbitrary number of messages.

To explore these design parameters in practice, we built two applications using Collage's message layer. The first

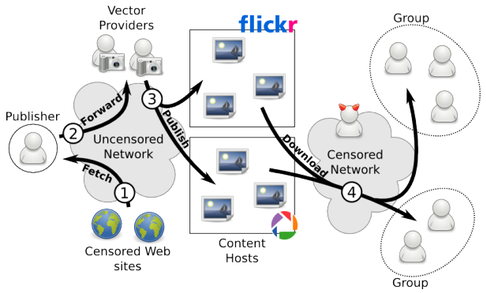



Figure 6: Proxied Web content passes through multiple parties before publication on content hosts. Each group downloads a different subset of images when fetching the same URL.

is a Web content proxy whose goal is to distribute content to many users; the second is a covert email system.

## 6.2 Web Content Proxy

We have built an asynchronous Web proxy using Collage's message layer, with which a publisher in an uncensored region makes content available to clients inside censored regimes. Unlike traditional proxies, our proxy shields both the identities of its users and the content hosted from the censor.

The proxy serves small Web documents, such as articles and blog posts, by steganographically encoding content into images hosted on photo-sharing websites like Flickr and Picasa. A standard steganography tool [38] can encode a few kilobytes in a typical image, meaning most hosted documents will fit within a few images. To host many documents simultaneously, however, the publisher needs a large supply of images; to meet this demand, the publisher operates a service allowing generous users of online image hosts to donate their images. The service takes the images, encodes them with message data, and returns the encoded images to their owners, who then upload them to the appropriate image hosts. Proxy users download these photos and decode their contents. Figure 6 summarizes this process. Notice that the publisher is outside the censored domain, which frees us from worrying about sender deniability.

To use a proxy, users must *discover* a publisher, *register* with that publisher, and be *notified* of an encryption key. Publishers are identified by their public key so discovering publishers is reduced to a key distribution exercise, albeit that these keys must be distributed without the suspicion of the censor. Several techniques are feasible: the key could be delivered alongside the client software, derived from a standard SSL key pair, or distributed offline. Like any key-based security system, our proxy must deal with this inherent bootstrapping problem.

Once the client knows the publisher's public key, it sends a message requesting registration. The message identifier is the publisher's public key and the message payload contains the public key of the client encrypted using the publisher's public key. This encryption ensures that only the publisher knows the client's public key. The publisher receives and decrypts the client's registration request using his own private key.

The client is now registered but doesn't know where content is located. Therefore, the publisher sends the client a message containing a *group key*, encrypted using the client's public key. The group key is shared between a small number of proxy users and is used to discover identifiers of content. For security, different groups of users fetch content from different locations; this prevents any one user from learning about (and attacking) all content available through the proxy.

After registration is complete, clients can retrieve content. To look up a URL $u$, a client hashes $u$ with a keyed hash function using the group key. It uses the hash as the message identifier for receive.

Unlike traditional Web proxies, only a limited amount of content is available though our proxy. Therefore, to accommodate clients' needs for unavailable content, clients can suggest content to be published. To suggest a URL, a client sends the publisher a message containing the requested URL. If the publisher follows the suggestion, then it publishes the URL for users of that client's group key.

Along with distributing content, the publisher provides updates to the task database via the proxy itself (at the URL `proxy://updates`). The clients occasionally fetch content from this URL to keep synchronized with the publisher's task database. The consistent hashing algorithm introduced in Section 4.3 allows updates to be relatively infrequent; by default, the proxy client updates its database when 20% of tasks have been remapped due to churn (*i.e.*, there is a 20% reduction in the number of successful task executions). Figure 4 shows that there may be many changes to the task database before this occurs.

**Implementation and Evaluation.** We have implemented a simple version of the proxy and can use it to publish and retrieve documents on Flickr. The task database is a set of tasks that search for combinations (*e.g.*, "vacation" and "beach") of the 130 most popular tags. A 23 KB one-day news summary requires nine JPEG photos ($\approx$ 3 KB data per photo, plus encoding overhead) and takes approximately 1 minute to retrieve over a fast network connection; rendering web pages and large photos takes a significant fraction of this time. Note that the document is retrieved immediately after publication; performance decays slightly over time because search results are displayed in reverse chronological order. We have also implemented a photo donation service, which accepts Flickr photos from users, encodes them with censored content, and uploads them on the user's behalf. This donation service is available for download [13].

## 6.3 Covert Email

Although our Web proxy provides censored content to many users, it is susceptible to attack from the censor for precisely this reason: because no access control is performed, the censor could learn the locations of published URLs using the proxy itself and potentially mount denial-of-service attacks. To provide greater security and availability, we present Covert Email, a point-to-point messaging system built on Collage's message layer that excludes the censor from sending or receiving messages, or observing its users. This design sacrifices scalability: to meet these security requirements, all key distribution is done out of band, similar to PGP key signing.

Messages sent with Covert Email will be smaller and potentially more frequent than for the proxy, so Covert Email uses text vectors instead of image vectors. Using text also improves deniability, because receivers are inside the censored domain, and publishing a lot of text (*e.g.*, comments, tweets) is considered more deniable than many photos. Blogs, Twitter, and comment posts can all be used to store message chunks. Because Covert Email is used between a closed group of users with a smaller volume of messages, the task database is smaller and updated less often without compromising deniability. Additionally, users can supply the text vectors needed to encode content (*i.e.*, write or generate them), eliminating the need for an outside vector source. This simplifies the design.

Suppose a group of mutually trusted users wishes to communicate using Covert Email. Before doing so, it must establish a shared secret key, for deriving message identifiers for sending and receiving messages. This one-time exchange is done out-of-band; any exchange mechanism works as long as the censor is unaware that a key exchange takes place. Along with exchanging keys, the group establishes a task database. At present, a database is distributed with the application; the group can augment its task database and notify members of changes using Covert Email itself.

Once the group has established a shared key and a task database, its members can communicate. To send email to Bob, Alice generates a message identifier by encrypting a tuple of his email address and the current date, using the shared secret key. The date serves as a salt and ensures variation in message locations over time. Alice then sends her message to Bob using that identifier. Here, Bob's email address is used only to uniquely identify him within the group; in particular, the domain portion of the address serves no purpose for communication within the group.

To receive new mail, Bob attempts to receive messages with identifiers that are the encryption of his email address and some date. To check for new messages, he checks each date since the last time he checked mail. For example, if Bob last checked his mail yesterday, he checks two dates: yesterday and today.

If one group member is outside the censored domain, then *Covert Email can interface with traditional email.* This user runs an email server and acts as a proxy for the other members of the group. To send mail, group members send a message to the proxy, requesting that it be forwarded to a traditional email address. Likewise, when the proxy receives a traditional email message, it forwards it to the requisite Covert Email user. This imposes one obvious requirement on group members sending mail using the proxy: they must use email addresses where the domain portion matches the domain of the proxy email server. Because the domain serves no other purpose in Covert Email addresses, implementing this requirement is easy.

**Implementation and Evaluation.** We have implemented a prototype application for sending and retrieving Covert Email. Currently, the task database is a set of tasks that search posts of other Twitter users. We have also written tasks that search for popular keywords (*e.g.*, "World Cup"). To demonstrate the general approach, we have implemented an (insecure) proof-of-concept steganography algorithm that stores data by altering the capitalization of words. Sending a short 194-byte message required three tweets and took five seconds. We have shown that Covert E-mail has the potential to work in practice, although this application obviously needs many enhancements before general use, most notably a secure text vector encoding algorithm and more deniable task database.

# 7 Threats to Collage

This section discusses limitations of Collage in terms of the security threats it is likely to face from censors; we also discuss possible defenses. Recall from Section 3.2 that we are concerned with two security metrics: *availability* and *deniability*. Given the unknown power of the censor and lack of formal information hiding primitives in this context, both goals are necessarily best effort.

### 7.1 Availability

A censor may try to prevent clients from sending and receiving messages. Our strongest argument for Collage's availability depends on a censor's unwillingness to block large quantities of legitimate content. This section discusses additional factors that contribute to Collage's current and future availability.

The censor could *block message vectors*, but a censor that wishes to allow access to legitimate content may have trouble doing so since censored messages are encoded inside otherwise legitimate content, and message vectors are, by design, difficult to remove without destroying the cover content. Furthermore, some encoding schemes (*e.g.*, steganography) are resilient against more determined censors, because they hide the presence of Collage data; blocking encoded vectors then also requires blocking many legitimate vectors.

The censor might instead *block traffic patterns resembling Collage's tasks*. From the censor's perspective, doing so may allow legitimate users access to content as long as they do not use one of the many tasks in the task database to retrieve the content. Because tasks in the database are "popular" among innocuous users by design, blocking a task may also disrupt the activities of legitimate users. Furthermore, if applications prevent the censor from knowing the task database, mounting this attack becomes quite difficult.

The censor could *block access to content hosts*, thereby blocking access to vectors published on those hosts. Censors have mounted this attack in practice; for example, China is currently blocking Flickr and Twitter, at least in part [43]. Although Collage cannot prevent these sites from being blocked, applications can reduce the impact of this action by publishing vectors *across many user-generated content sites*, so even if the censor blocks a few popular sites there will still be plenty of sites that host message vectors. One of the strengths of Collage's design is that it does not depend on any specific user-generated content service: any site that can host content for users can act as a Collage drop site.

The censor could also try to *prevent senders from publishing content*. This action is irrelevant for applications that perform all publication outside a censored domain. For others, it is impractical for the same reasons that blocking receivers is impractical. Many content hosts (*e.g.*, Flickr, Twitter) have third-party publication tools that act as proxies to the publication mechanism [51]. Blocking all such tools is difficult, as evidenced by Iran's failed attempts to block Twitter [14].

Instead of blocking access to publication or retrieval of user-generated content, the censor could *coerce content hosts* to remove vectors or disrupt the content inside them. For certain vector encodings (*e.g.*, steganography) the content host may be unable to identify vectors containing Collage content; in other cases (*e.g.*, digital watermarking), removing encoded content is difficult without destroying the outward appearance of the vector (*e.g.*, removing the watermark could induce artifacts in a photograph).

### 7.2 Deniability

As mentioned in Section 3.1, the censor may try to compromise the deniability of Collage users. Intuitively, a Collage user's actions are *deniable* if the censor cannot distinguish the use of Collage from "normal" Internet activity. Deniability is difficult to quantify; others have developed metrics for anonymity [39], and we are working on quantitative metrics for deniability in our ongoing work. Instead, we explore deniability somewhat more informally and aim to understand how a censor can attack a Collage user's deniability and how future extensions to Collage might mitigate these threats. The censor may attempt to compromise the deniability of either the sender or the receiver of a message. We explore various ways the censor might mount these attacks, and possible extensions to Collage to defend against them.

The censor may attempt to **identify senders**. Applications can use several techniques to improve deniability. First, they can *choose deniable content hosts*; if a user has never visited a particular content host, it would be unwise to upload lots of content there. Second, *vectors must match tasks*; if a task requires vectors with certain properties (*e.g.*, tagged with "vacation"), vectors not meeting those requirements are not deniable. The vector provider for each application is responsible for ensuring this. Finally, *publication frequency must be indistinguishable* from a user's normal behavior and the publication frequency of innocuous users.

The censor may also attempt to **identify receivers**, by observing their task sequences. Several application parameters affect receiver deniability. As the *size of the task database* grows, clients have more variety (and thus deniability), but must crawl through more data to find message chunks. Increasing the *number of tasks mapped to each identifier* gives senders more choice of publication locations, but forces receivers to sift through more content when retrieving messages. Increasing *variety of tasks* increases deniability, but requires a human author to specify each type of task. The receiver must decide an *ordering of tasks* to visit; ideally, receivers only visit tasks that are popular among innocuous users.

Ultimately, the censor may develop more sophisticated techniques to defeat user deniability. For example, a censor may try to target individual users by mounting timing attacks (as have been mounted against other systems like Tor [4, 33]), or may look at how browsing patters change across groups of users or content sites. In these cases, we believe it is possible to extend Collage so that its request patterns more closely resemble those of innocuous users. To do so, Collage could monitor a user's normal Web traffic and allow Collage traffic to only perturb observable distributions (*e.g.*, inter-request timings, traffic per day, etc.) by small amounts. Doing so could obviously have massive a impact on Collage's performance. Preliminary analysis shows that over time this technique could yield sufficient bandwidth for productive communication, but we leave its implementation to future work.

## 8 Conclusion

Internet users in many countries need safe, robust mechanisms to publish content and the ability to send or publish messages in the face of censorship. Existing mechanisms for bypassing censorship firewalls typically rely on establishing and maintaining infrastructure outside the censored regime, typically in the form of proxies; unfortunately, when a censor blocks these proxies, the systems are no longer usable. This paper presented Collage, which bypasses censorship firewalls by piggybacking messages on the vast amount and types of user-generated content on the Internet today. Collage focuses on providing both availability and some level of deniability to its users, in addition to more conventional security properties.

Collage is a further step in the ongoing arms race to circumvent censorship. As we discussed, it is likely that, upon seeing Collage, censors will take the next steps towards disrupting communications channels through the firewall—perhaps by mangling content, analyzing joint distributions of access patterns, or analyzing request timing distributions. However, as Bellovin points out: "There's no doubt that China—or any government so-minded—can censor virtually everything; it's just that the cost—cutting most communications lines, and deploying enough agents to vet the rest—is prohibitive. The more interesting question is whether or not 'enough' censorship is affordable." [7] Although Collage itself may ultimately be disrupted or blocked, it represents another step in making censorship more costly to the censors; we believe that its underpinnings—the use of user-generated content to pass messages through censorship firewalls—will survive, even as censorship techniques grow increasingly more sophisticated.

## Acknowledgments

This work was funded by NSF CAREER Award CNS-0643974, an IBM Faculty Award, and a Sloan Fellowship. We thank our shepherd, Micah Sherr, and the anonymous reviewers for their valuable guidance and feedback. We also thank Hari Balakrishnan, Mike Freedman, Shuang Hao, Robert Lychev, Murtaza Motiwala, Anirudh Ramachandran, Srikanth Sundaresan, Valas Valancius, and Ellen Zegura for feedback.

## References

[1] Selenium Web application testing system. `http://www.seleniumhq.org`.

[2] Riaa sues computer-less family, 234 others, for file sharing. `http://arstechnica.com/old/content/2006/04/6662.ars`, apr 2006.

[3] Anonymizer. `http://www.anonymizer.com/`.

[4] A. Back, U. Möller, and A. Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In I. S. Moskowitz, editor, *Proceedings of Information Hiding Workshop (IH 2001)*, pages 245–257. Springer-Verlag, LNCS 2137, April 2001.

[5] A. Baliga, J. Kilian, and L. Iftode. A web based covert file system. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[6] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker. Low-resource routing attacks against tor. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2007)*, Washington, DC, USA, Oct. 2007.

[7] S. M. Bellovin. A Matter of Cost. *New York Times* Room for Debate Blog. Can Google Beat China? `http://roomfordebate.blogs.nytimes.com/2010/01/15/can-google-beat-china/#steven`, Jan. 2010.

[8] P. Boucher, A. Shostack, and I. Goldberg. Freedom systems 2.0 architecture. White paper, Zero Knowledge Systems, Inc., December 2000.

[9] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proc. ACM SIGCOMM*, pages 56–67, Vancouver, British Columbia, Canada, Sept. 1998.

[10] China Web Sites Seeking Users' Names. `http://www.nytimes.com/2009/09/06/world/asia/06chinanet.html`, Sept. 2009.

[11] Chinese blogger Xu Lai stabbed in Beijing bookshop. `http://www.guardian.co.uk/world/2009/feb/15/china-blogger-xu-lai-stabbed`, Feb. 2009.

[12] I. Clarke. A distributed decentralised information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.

[13] Collage. `http://www.gtnoise.net/collage/`.

[14] Could Iran Shut Down Twitter? `http://futureoftheinternet.org/could-iran-shut-down-twitter`, June 2009.

[15] G. Danezis. Covert communications despite traffic data retention.

[16] G. Danezis and C. Diaz. A survey of anonymous communication channels. Technical Report MSR-TR-2008-35, Microsoft Research, January 2008.

[17] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 2–15, May 2003.

[18] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proc. 13th USENIX Security Symposium*, San Diego, CA, Aug. 2004.
[19] China is number one. *The Economist*, Jan. 2009. http://www.economist.com/daily/chartgallery/displaystory.cfm?story_id=13007996.
[20] Facebook. http://www.facebook.com/.
[21] N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan, and D. Karger. Infranet: Circumventing Web censorship and surveillance. In *Proc. 11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.
[22] N. Feamster, M. Balazinska, W. Wang, H. Balakrishnan, and D. Karger. Thwarting Web censorship with untrusted messenger discovery. In *3rd Workshop on Privacy Enhancing Technologies*, Dresden, Germany, Mar. 2003.
[23] N. Feamster and R. Dingledine. Location diversity in anonymity networks. In *ACM Workshop on Privacy in the Electronic Society*, Washington, DC, Oct. 2004.
[24] Flickr. http://www.flickr.com/.
[25] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proc. 9th ACM Conference on Computer and Communications Security*, Washington, D.C., Nov. 2002.
[26] Freedom on the Net. Technical report, Freedom House, Mar. 2009. http://www.freedomhouse.org/uploads/specialreports/NetFreedom2009/FreedomOnTheNet_FullReport.pdf.
[27] J. Fridrich, M. Goljan, and D. Hogea. Attacking the outguess. In *Proceedings of the ACM Workshop on Multimedia and Security*, 2002.
[28] Future of Open Source: Collaborative Culture. http://www.wired.com/dualperspectives/article/news/2009/06/dp_opensource_wired0616, June 2009.
[29] A. Hintz. Fingerprinting websites using traffic analysis. In *Workshop on Privacy Enhancing Technologies*, San Francisco, CA, Apr. 2002.
[30] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.
[31] South Korea mulls Web watch , June 2008. http://www.theinquirer.net/inquirer/news/091/1042091/south-korea-mulls-web-watch.
[32] P. Maymounkov. Online codes. Technical Report TR2002-833, New York University, Nov. 2002.
[33] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*. IEEE CS, May 2005.
[34] Cisco netflow. http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.
[35] Uproar in Australia Over Plan to Block Web Sites, Dec. 2008. http://www.nytimes.com/aponline/2008/12/26/technology/AP-TEC-Australia-Internet-Filter.html?_r=1.
[36] OpenNet Initiative. http://www.opennet.net/.
[37] Report on china's filtering practices, 2008. Open Net Initiative. http://opennet.net/sites/opennet.net/files/china.pdf.
[38] Outguess. http://www.outguess.org/.
[39] A. Serjantov and G. Danezis. Towards an information theoretic metric for anonymity. In R. Dingledine and P. Syverson, editors, *Proceedings of Privacy Enhancing Technologies Workshop (PET 2002)*. Springer-Verlag, LNCS 2482, April 2002.
[40] A. Serjantov and P. Sewell. Passive attack analysis for connection-based anonymity systems. In *Proceedings of ESORICS 2003*, Oct. 2003.
[41] The SNOW Home Page. http://www.darkside.com.au/snow/.
[42] Y. Sovran, J. Li, and L. Subramanian. Pass it on: Social networks stymie censors. In *Proceedings of the 7th International Workshop on Peer-to-Peer Systems*, Feb. 2008.
[43] TechCrunch. China Blocks Access To Twitter, Facebook After Riots. http://www.techcrunch.com/2009/07/07/china-blocks-access-to-twitter-facebook-after-riots/.
[44] Tor: Bridges. http://www.torproject.org/bridges.
[45] Tor partially blocked in China, Sept. 2009. https://blog.torproject.org/blog/tor-partially-blocked-china.
[46] TorrentFreak. China Hijacks Popular BitTorrent Sites. http://torrentfreak.com/china-hijacks-popular-bittorrent-sites-081108/, May 2008.
[47] Pakistan move knocked out YouTube, Jan. 2008. http://www.cnn.com/2008/WORLD/asiapcf/02/25/pakistan.youtube/index.html.
[48] Turkey blocks YouTube access, Jan. 2008. http://www.cnn.com/2008/WORLD/europe/03/13/turkey.youtube.ap/index.html.
[49] Twitter. http://www.twitter.com.
[50] 18 Million Twitter Users by End of 2009. http://mashable.com/2009/09/14/twitter-2009-stats/, Sept. 2009.
[51] Ultimate List of Twitter Applications. http://techie-buzz.com/twitter/ultimate-list-of-twitter-applications-and-websites.html, 2009.
[52] State of the Twittersphere. http://bit.ly/sotwitter, 2009.
[53] M. Waldman and D. Mazières. Tangler: A censorship-resistant publishing system based on document entanglements. In *Proc. 8th ACM Conference on Computer and Communications Security*, Philadelphia, PA, Nov. 2001.
[54] The Accidental Censor: UK ISP Blocks Wayback Machine, Jan. 2009. Ars Technica. http://tinyurl.com/dk7mhl.
[55] Wikipedia, Cleanfeed & Filtering, Dec. 2008. http://www.nartv.org/2008/12/08/wikipedia-cleanfeed-filtering.
[56] Youtube - broadcast yourself. http://www.youtube.com/.
[57] YouTube Statistics. http://ksudigg.wetpaint.com/page/YouTube+Statistics, Mar. 2008.



# Fighting Coercion Attacks in Key Generation using Skin Conductance

Payas Gupta
*School of Information Systems*
*Singapore Management University*
payas.gupta.2008@phdis.smu.edu.sg

Debin Gao
*School of Information Systems*
*Singapore Management University*
dbgao@smu.edu.sg

## Abstract

Many techniques have been proposed to generate keys including text passwords, graphical passwords, biometric data and etc. Most of these techniques are not resistant to coercion attacks in which the user is forcefully asked by an attacker to generate the key to gain access to the system or to decrypt the encrypted file. We present a novel approach in generating cryptographic keys to fight against coercion attacks. Our novel technique incorporates the user's emotional status, which changes when the user is under coercion, into the key generation through measurements of the user's skin conductance. We present a model that generates cryptographic keys with one's voice and skin conductance. In order to explore more, a preliminary user study with 39 subjects was done which shows that our approach has moderate false-positive and false-negative rates. We also present the attacker's strategy in guessing the cryptographic keys, and show that the resulting change in the password space under such attacks is small.

## 1 Introduction

Many techniques have been proposed to generate strong cryptographic keys for secure communication and authentication. Some of these techniques, e.g., those using biometrics [15, 24, 27, 28, 35], offer desirable security properties including ease of use, unforgettability, unforgeability (to some extent), high entropy and etc. However, most of these schemes are not resistant to coercion attacks in which the user is forcefully asked by an attacker to reveal the key [32]. When the user's life is threatened by an attacker, one would have to surrender the key, and the system will be compromised despite all the security properties described above. In this paper, we present a novel approach to protection against coercion attacks in generating keys.

For a cryptographic key generation technique to be coercion attack resistant, it is required that when the user is under coercion, he/she will have no way of generating the key, or the key generated will never be the same as the one generated when he/she is not being coerced. If this requirement is met, then an adversary would not apply any threat to him/her because the adversary understands that the user would not be able to generate the key when he is threatened to do so. Here we assume that the coercion resistance property is publicly known to everyone, including the attackers; otherwise it might lead to a dangerous situation for the user, a problem we do not address in this paper.

To show how desirable it is to have a coercion-resistant cryptographic key generation technique, here we list a few scenarios in which such a technique could be useful:

- Bank's vault and safe: According to statistics released by the FBI [17], there were $1,094$ reported robberies (out of which 58 cases were of vault/safe robberies) of commercial banks between July 1, 2009 and September 30, 2009 totaling more than $9.4 million. If such systems are used to fight against these attacks, then managers will never be forced to open the vault.
- Cockpit doors on airliners: The hijackers of the September 11, 2001 use the fueled aircraft as a missile to destroy ground targets. If the cockpit doors on airliners are well equipped with coercion resisted techniques, then hijackers can never force a flight attendant to open the door.
- Secret/capability holders in a war: secret and capability holders would not be forced to reveal the secret or use the capability.

In this paper, we explore the incorporation of user's emotional status (through the measure of skin conductance) into the process of key generation to achieve coercion resistance. We demonstrate this possibility by incorporating skin conductance into a previously proposed

key generation technique using biometrics [24] (see Figure 1).

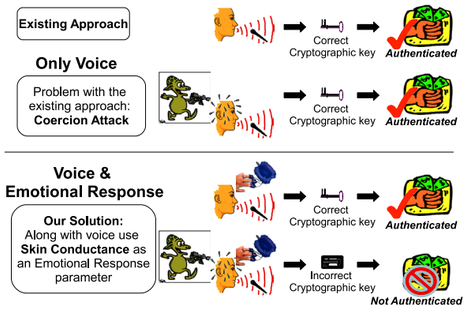



Figure 1: Coercion attacks in key generation

Incorporating skin conductance information into key generation is nontrivial. First, the fact that a change in a user's emotional status leads to changes in a user's skin conductance does not necessarily mean that our proposed technique is coercion resistant. If known patterns exist in such changes, an attacker might be able to guess the skin conductance of the user when he is not nervous by, e.g., flipping a few bits of the feature key (see Section 4) generated from the skin conductance of the user when he is nervous. We analyze this attack and its consequences, and show that the reduction in password space is small.

Second, we hope that the key generation algorithm will take in the least amount of user specific information except the live data collected when it is used. This is because the key generation algorithm might be executed from the client's machine, and the inputs to the algorithm could potentially be retrieved by the attacker during a coercion attack. However, when dealing with biometrics data, removing such user specific information from the inputs of the algorithm is not plausible, as different people have different sets of consistent and inconsistent biometric features. The algorithm would have too high false-negative rates without this additional user specific information. We propose using only user-specific feature lookup tables which contain valid key shares or garbage. We also analyze conceivable attacks that result from our proposal.

Third, it is nontrivial how a user study can be performed to evaluate our technique. We need to collect biometric data corresponding to different emotional states of real human beings. Efforts in this area are more demanding than traditional efforts to get pattern recognition data [31]. To analyze the effectiveness of our proposal, we perform a user study to see how one's skin conductance changes when he/she is being coerced. This is used to evaluate the false-positive and false-negative rates of our model, and to analyze the attacker's strategy in guessing the cryptographic key. With 39 participants in our user study, we find that our technique enjoys moderate false-positive and false-negative rates in key generation. Furthermore, we find that the reduction in the password space for an informed attacker is small.

The rest of the paper is organized as follows. In Section 2, we discuss some state-of-the-art approaches in cryptographic key generation and recognition of emotional response. Background knowledge about the chosen biometrics and fingerprint are discussed in Section 3. In Section 4, we present the details of our approach in key generation using skin conductance and voice. The user study and results are presented in Sections 5 and Section 6 respectively. We conclude in Section 7 with some plausible future work.

## 2 Related Work

In this section we review some of the techniques and methodologies used to generate cryptographic keys from biometrics and some previous work on the emotion recognition schemes using physiological signals.

Many key generation techniques from biometrics, e.g., voice, iris, face, fingerprints, keystroke dynamics, and etc., have been proposed in the last decade [15, 24, 27, 28, 35]. The pioneer work in cryptographic key generation from behavioral biometrics uses keystroke dynamics of a user while typing the password [25]. The features of interest are the duration of keystrokes and the latency between each pair of keystrokes. The generated cryptographic key is called the hardened password. However the password generated is not very long and is susceptible to brute-force attacks [25]. Another method using secret sharing was proposed to generate the biometric key from voice [24]. The distinguishing biometric features are selected based on the separation between the authentic and the imposter data, and then binarized by some thresholds. However, this method is not resistant to coercion attacks (which our proposed model trying to target), as the attacker can force the user to speak out the password in a normal way. We will discuss key generation approach from voice in more detail in the formal framework of our model (see Section 3).

Another work on key generation from voice uses phonemes instead of words, as it is possible to generate larger keys with shorter sequences [15]. Using the information of the voice model and the phoneme information of the segments, a set of features are created to train an SVM (Support Vector Machine) that could generate a cryptographic key. False-positives and entropy of the system were not demonstrated, which does not give a clear picture of the security of the scheme.



There are many risk and security concerns over biometric systems [32, 33, 40]. Some of the threat models include fake biometrics at the sensor, tampering with the stored templates, coercion attacks. Biometrics liveness detection is proposed to thwart fake biometrics attacks, e.g., by using perspiration in the skin [1] or blood flow [22]. However, no previous work has been proposed to resist coercion attacks in generating cryptographic keys using biometrics. There have been suggestions like panic alarm or duress code to fight against coercion attacks, but they are different from what we are proposing here because in previous schemes users *choose* not to generate the key but to send a signal to authorities without catching the adversary's attention, whereas in our scheme we require that users simply will not be able to generate the key. It is clear that our scheme offers much stronger security properties.

Previous work also shows that emotion recognition using physiological signals, affects from speech, and facial expressions have various success rates between 60% and 98% [31]. Although many techniques have been proposed for emotion recognition [31, 20, 29, 21], none has looked into the incorporation of emotional status into key generation as what we propose in this paper.

## 3 Background

In this section, we present some background knowledge of voice and skin conductance, and discuss why in future an addition of fingerprint in our model would be better as an authentication measure for the protection against coercion attack. We also discuss the reasons for the selection of these features and the advantages over others in terms of acceptability, feasibility and usability.

### 3.1 Why Skin Conductance?

An emotion is a mental and physiological state associated with a wide variety of feelings, thoughts, and behavior. Emotions are subjective experiences, often associated with mood, temperament, personality, and disposition [11]. This emotional behavioral change is the key component in our model in fighting against coercion attack. Several physiological peripheral activities have been found to be related to emotional processing of situations. Many physiological parameters were studied for emotion recognition, e.g., heart beat rate [3] (HR), skin conductance [23] (SC), EMG (Electromyography) signals, ECG (Electrocardiography) signals, body temperature, BVP (Blood Volume Pulse) signals, and etc., among which HR and SC are especially attractive due to their strong association with behavioral activation system (BAS) and behavioral inhibition system (BIS) respectively [14].

SC is the change in the electrical properties of an individual person's skin caused by an interaction between environmental events and the individual psychological state. Human skin is a good conductor of electricity and when subject to a weak electrical current, a change in the skin conductance level occurs [42]. We chose SC over HR for the following reasons.

1. The skin conductance is one of the fastest responding measures of stress response [16]. It is one of the most robust and non-invasive physiological measures of autonomic nervous system activity [7]. Researchers have linked skin conductance response to stress and autonomic nervous system arousal [37].

2. The change in HR not only accounts for stress but for many other reasons, including jogging or doing some heavy work load. SC, on the other hand, has been shown to be a promising measure in experimental studies [36] for its reliability.

3. According to [41], HR is also impacted when stress levels rise but the shifts take a bit of time to happen and by the time the changes are noticeable the triggering stimulus is long past, whereas SC responses are rapid and easy to measure.

4. HR is not suitable to our model due to prevailing feasibility issues. HR can be measured using an Electrocardiogram (ECG) machine or a stethoscope. Using an ECG machine is impractical because it is very cumbersome due to many (at least three) electrodes required and installation costs [6]. Stethoscope is not good either because different placements of the stethoscope could lead to high FTC rate (failure to capture rate) [30].

5. Using SC has an extra advantage as it can be measured simultaneously while fingerprints are being scanned. This ensures that SC is measured from the authentic person (more on this in the coming subsection). The wide acceptance of finger scanning [18, 39] also suggest that SC measurement would have the potential to gain user acceptance.

There are some limitations of using skin conductance as with any other biometric. Some skin lotions can be used to manipulate the skin conductance level. In a test done by [34], the usage of specific solutions produced significant increase in skin water content, and was indicated by increase in skin conductance level. According to the product after the application of the cream by EncoSkin, skin moisture level can be significantly increased which can be monitored by skin conductance [12].

### 3.2 Why Voice?

Voice has been used previously to generate cryptographic keys [15, 24]. Voice as a biometric is desirable for generating keys for two important reasons. First, it is the most familiar way of communication, which makes it ideal for many applications. Second, voice is a dynamic biometric and is not static like iris or fingerprint. A user can have different keys for different accounts by just changing the password (what to pronounce) or the vocalization of the same password (how to pronounce) to generate different cryptographic keys. In an event of key compromise a new cryptographic key can be easily generated. Note that voice has a potential disadvantage when used in fighting against coercion, namely that the attacker may blame the user for intentionally pronouncing the wrong password. We demonstrated our technique with voice; however, our scheme is not limited to using voice, other biometric can be used as well.

### 3.3 Why Fingerprint?

A potential threat to our biometric system is to use spoken password from the genuine user *(under stress)* and SC responses from another person *(normal emotional state)*. To ensure that SC is not unforgeable, one can make use of a device to collect fingerprint and skin conductance of the user at the same time so that the fingerprint of the user can be checked and mapped to his/her skin conductance signal. However, we did not demonstrate how to use this as a measure in our proposed model as this is not the contribution of this paper and is left for the future work.

## 4 Key Generation from Voice and Skin Conductance

In order to show how skin conductance can be used to fight against coercion attacks in cryptographic key generation, in this section, we present the details of a cryptographic key generation technique using voice and skin conductance. Note the criteria behind choosing skin conductance and voice in Section 3. Other biometrics in lieu of voice could be used as well. Our way of using voice is similar (with some differences) to an earlier proposal of generating cryptographic keys using voice [24]. Table 1 shows some notations used in the rest of this paper.

### 4.1 An Overview

Inputs to our model include the voice captured when the user utter the password into the microphone and the skin conductance measured. Figure 2 shows the input devices

Figure 2: Input devices

we used in our experimental setup. Output of our model is a cryptographic key generated.

In the first phase (Figure 3 (a)–(h)), features extracted from the spoken password are used to generate a sequence of frames $f_V(1), \ldots, f_V(n)$ (3 (c)), from which an optimal segmentation of $s$ segments (component sounds) (3 (f)). The segmentation obtained are then mapped to the feature descriptor using a random $\alpha_V$ plane (3 (g)). Furthermore, features are also extracted from the SC sample and the corresponding feature descriptors are computed (3 (h)). These feature descriptors should be "sufficiently similar" for the same user and "sufficiently different" for different users. By the end of the first phase, we have feature descriptors for both voice and SC signal.

In the second phase (Figure 3 (i)–(l)), we perform lookup table generation and cryptographic key reconstruction. A total of $N_V$ samples from voice and $N_{SC}$ samples from SC are used to generate lookup tables $T_V$ and $T_{SC}$ . In cryptographic key reconstruction, feature keys are generated from the spoken password ($m_V$ bits) and SC ($m_{SC}$ bits). The two lookup tables generated and the features keys are then used to generate the cryptographic key.

In the next two subsections, we will present these two phases in more detail.

### 4.2 Phase I: Feature descriptors derivation

#### 4.2.1 Feature descriptors from voice

In the last six decades, speech recognition and speaker recognition have advanced a lot [8]. A speaker recognition system usually has three modules: feature extraction, pattern matching and decision making, among which feature extraction is especially important to our research as it estimates a set of features from the speech signal that represent the speaker-specific information. These features should be consistent for each speaker and should not change over time. The way we extract these features and derive the feature descriptors is very similar to the previous approach [24], except that we use the

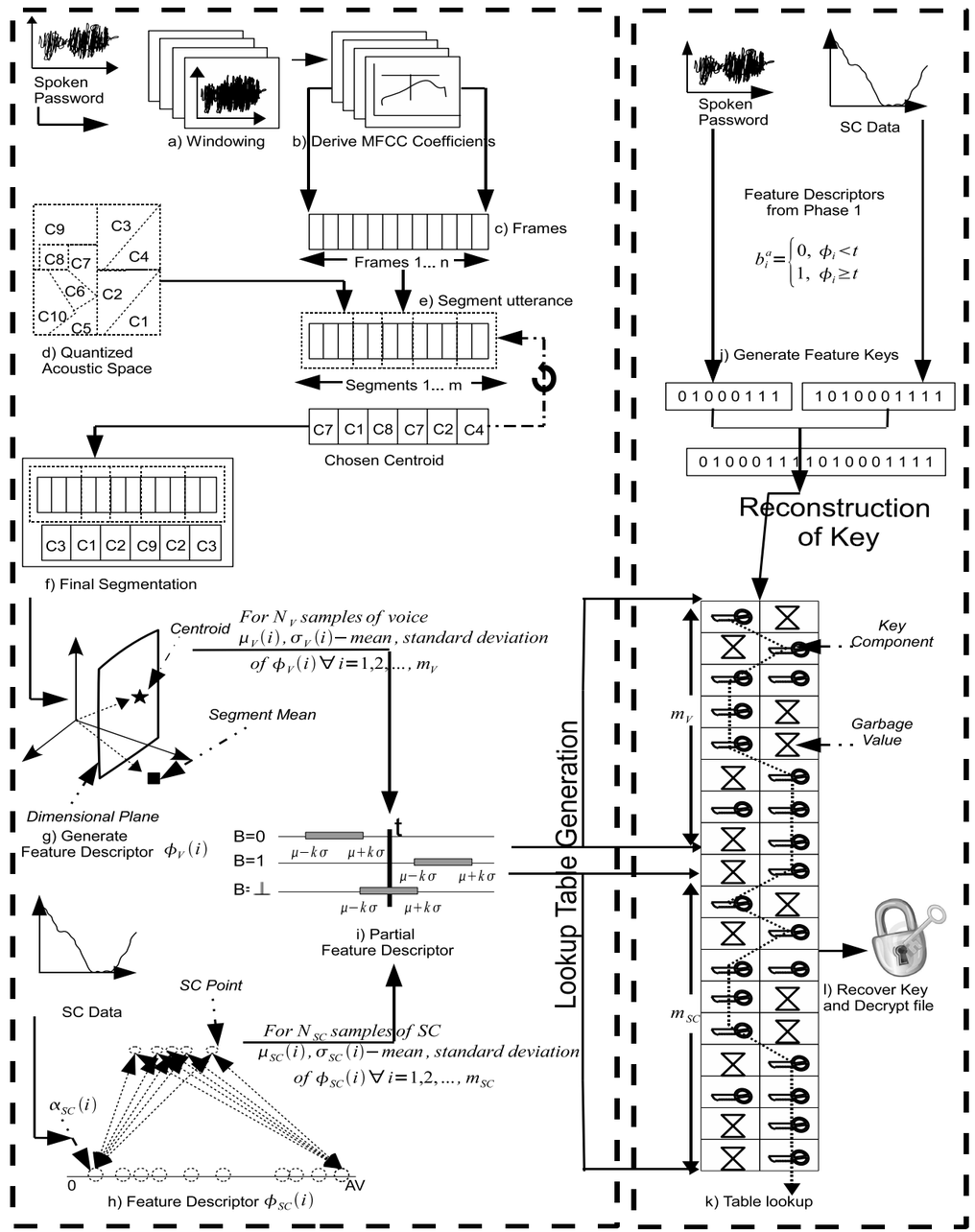



Figure 3: Design overview, refer to Section 4.2 for detailed description

| General Notations | | Notations related to Spoken Password | | Notations related to Skin Conductance | |
|---|---|---|---|---|---|
| $\mathcal{K}$ | cryptographic key | V | Voice | SC | Skin Conductance |
| $C$ | a set of centroids | $N_V$ | # samples in V during training | $N_{SC}$ | # samples in SC during training |
| $c$ | a centroid in $C$ | $f_V$ | frame vector | $f_{SC}$ | vector containing sampled values of SC |
| $m$ | $m = m_V + m_{SC}$ | $\phi_V$ | feature descriptor | $\phi_{SC}$ | feature descriptor |
| | | $n$ | number of frames | $\ell$ | number of frames |
| | | $T_V$ | lookup table generated using V | $T_{SC}$ | lookup table generated using SC |
| | | $m_V$ | total bits in a feature descriptor of V | $m_{SC}$ | total bits in a feature descriptor of SC |
| | | $b_V$ | feature key using V | $b_{SC}$ | feature key using SC |
| | | $s$ | number of segments | | |
| | | $R$ | segment vector | | |

Table 1: Notations

Mel-frequency Cepstral Coefficients (MFCCs) instead of linear cepstrum [24]. MFCC has advantages over linear cepstrum that the frequency bands are equally spaced on the mel scale, which approximates the human auditory system's response more closely than the linearly-spaced frequency bands used in the linear cepstrum [13].

**Associating centroids to the acoustic model** We convert the raw speech signal into a sequence of acoustic feature vectors in terms of the Mel-frequency Cepstral Coefficients (MFCCs) [10]. In the next paragraph we provide a short description on the extraction of MFCC (see Figure 4).

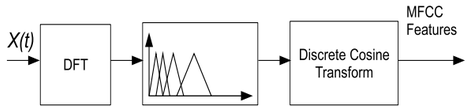



Figure 4: Block diagram of extracting MFCC

The voice signal is first divided into blocks of 20 to 30 msec (see Figure 3(a)), and Discrete Fourier Transform (DFT) is performed to obtain the frequency representation of each block. The neighboring frequencies in each block are grouped into bins of overlapping triangular bands of equal bandwidth. These bins are equally spaced on a Mel-scale instead of a normal scale as the lower frequencies are perceptually more important than the higher frequencies. The content of each band is now summed and the logarithmic of each sum is computed. To see this effect in time domain, Discrete Cosine Transform is applied to yield a "spectrum like" representation $\psi(t)$ that collectively make up an *MFC*, and $\psi(1), \ldots \psi(12)$ are called MFCC, where higher order coefficients are discarded. This vector is called a frame ($f_V$).

We run a sliding window of 30 msec over an utterance to obtain blocks 10 msec apart from one another, and extract the MFCC, $\langle \psi(1), \ldots \psi(12) \rangle$, for each block (see Figure 3(b)). $n$ frames are obtained from utterance of the password (see Figure 3(c)). An acoustic model of vectors from a speaker-independent and text-independent database of voice signals is obtained, from which vector quantization is used to partition the acoustic model into clusters (see Figure 3(d)). A multivariate normal distribution for each cluster is generated, where each cluster is parameterized by the vector $c$ of a component-wise means (called a centroid) and the covariance matrix $\Sigma$ for the vectors in the cluster. The density function for this distribution is

$$P(c \mid x) = \frac{1}{(2\pi)^{\delta/2}\sqrt{det(\Sigma)}} e^{-(x-c)^T \Sigma^{-1}(x-c)/2}$$

where $\delta$ is the dimension of the vectors. We denote the set of centroids as $C$.

**Segmentation of frames** After getting the centroids from a speaker-independent database of voice signals, we try to obtain the transcription, i.e., the starts and ends, of the phonemes of an individual user's utterance.

To do this, we perform segmentation on the spoken password. Let $f_V(1), \ldots f_V(n)$ be the sequence of frames from the utterance, and $F(R_1), \ldots F(R_s)$ be the sequence of $s$ segments ($s$ is a constant and same for all users), where $F(R_i)$ is the $i^{th}$ segment containing the sequence of frames $f_V(j), \ldots f_V(j')$ such that, $1 \leq j \leq j' \leq n$. Intuitively, each $F(R_i)$ corresponds to one "component sound" of the user's utterance.

We did this with an iterative approach (see algorithm 1). Ranges $R_1, \ldots, R_s$ are first initialized to be equally long. We then calculate the matching centroid $c$ for a segment F($R$), i.e., the one for which the likelihood of F($R$) w.r.t. $c$ is maximum. Dynamic programming is then used to determine a new segmentation for that frame sequence. This process is repeated until an optimal segmentation is obtained, which is mapped to the feature descriptor (see Figure 3(e,f)).

**Feature descriptor** Having derived a segmentation for a spoken password, we next define the feature descriptor ($\phi_V$) of this segmentation that is typically the same when the same user speaks out the same utterance. To do this,



**Algorithm 1** Spoken password segmentation

Segmentation $(f_V(1), \ldots, f_V(n), s)$

Score$'$ $\longleftarrow 0$
**for** $i = 1$ to $s$ **do**
  $R_i \longleftarrow \left( \left\lfloor \frac{(i-1) \times n}{s} \right\rfloor, \left\lfloor \frac{i \times n}{s} \right\rfloor \right)$
**end for**
**repeat**
  Score $\longleftarrow$ Score$'$
  **for** $i = 1$ to $s$ **do**
    **while** $\forall c \in C$ **do**
      $L(F(R_i)|c) \longleftarrow \prod_{j \in R_i} (f_V(j)|c)$
    **end while**
    $c(R_i) \longleftarrow arg \max_{c \in C} \{L(F(R)|c)\}$
  **end for**
  let $\bigcup_{i=1}^{s} R_i' \longleftarrow [1, n]$
  Score$'$ $\longleftarrow \prod_{i=1}^{s} L(F(R_i')|c(R_i)))$
  $R_i \longleftarrow R_i'$
**until** Score$'$ - Score $< \Delta$

we use a fixed vector $\alpha_V$, and define the $i^{th}$ bit of the feature descriptor as (see Figure 3(g))

$$\phi_V(i) = \alpha_V.(\mu_V(R_i) - c(R_i)), \quad \forall \quad 1 \leq i \leq s$$

That is, we normalize $\mu_V(R_i)$ with $c(R_i)$ and let $\phi_V$(i) be the linear combination of components in it as specified by $\alpha_V$. This process results in a feature descriptor ($\phi_V$), where $N_V$ feature descriptors are then generated from $N_V$ voice samples and used to generate a lookup table $T_V$ (in Phase II).

#### 4.2.2 Feature descriptor from skin conductance

When some external or internal stimuli occur that makes a person stressed, the skin becomes a better conductor of electricity. This conductance can be measured between two points on the body (e.g., two fingers) and the level of electrical conductance is called skin conductance. Since we want to detect changes in the emotional status of a person, we record skin conductance over a time period.

SC signal was measured with our device and sampled at a frequency of 30 samples per second. Let $f_{SC}(1), \ldots, f_{SC}(\ell)$ denote the sampled values obtained from the SC signal. We model the feature values into a feature descriptor ($\phi_{SC}$) in a similar way as we did in the processing of voice. We choose a random vector $\alpha_{SC}$= $[\alpha_{SC}(1), \alpha_{SC}(2), \ldots, \alpha_{SC}(m_{SC})]$ ($m_{SC}$ is a constant), and use the Euclidean distance between all the points of the $\alpha_{SC}$ vector and $f_{SC}$ to compute the distance measure M and henceforth the feature descriptor ($\phi_{SC}$).

$$M(i,j) = \alpha_{SC}(i) \times f_{SC}(j) \quad \forall \quad 1 \leq i \leq m_{SC}, 1 \leq j \leq \ell$$

$\phi_{SC}$ is the mean of all the distance measures for each $\alpha_{SC}(i)$ values (see Figure 3(h)), i.e.,

$$\phi_{SC}(i) = \frac{1}{\ell} \sum_{j=1}^{\ell} M(i,j) \quad \forall \quad 1 \leq i \leq m_{SC}$$

Note that the upper bound of $\alpha_{SC}(i)$ needs to be carefully chosen to maintain a good entropy on the feature descriptor of different people. Also note that we do not store skin conductance information directly but rather the feature descriptor generated from the distance measure is stored (same as in the case of voice). $N_{SC}$ feature descriptors are derived from $N_{SC}$ SC samples and then are used to generate a lookup table $T_{SC}$ (in Phase II).

### 4.3 Phase II: Lookup table and cryptographic key generation

We explain how we obtained the feature descriptors from voice and skin conductance in the previous subsection. Here, we will explain how we constructed lookup tables (training of the model) and obtained the cryptographic keys from the tables (usage of the model). The basic idea is that each entry of the lookup tables contains a share of the correct key or some garbage value, and the feature descriptor is used to determine the corresponding entry from the lookup table. In the end, the shares from the lookup tables are used to reconstruct the key.

#### 4.3.1 Lookup table generation

Intuitively, if a feature descriptor is the same as the one recorded previously (i.e., in training), then the system should choose the correct key share from the lookup table, or the garbage otherwise. In order to tolerate some small deviation of a user's utterance and skin conductance, we calculate the mean ($\mu_{\phi_V}$(i), $\mu_{\phi_{SC}}$(i)) and standard deviation ($\sigma_{\phi_V}$(i), $\sigma_{\phi_{SC}}$(i)) of each feature descriptor over $N_V$, $N_{SC}$ training samples, and define the partial feature descriptors $B_V$, $B_{SC}$ as

$$B_V(i) = \begin{cases} 0, & \text{if } \mu_{\phi_V}(\text{i}) + k\sigma_{\phi_V}(\text{i}) < t_V \\ 1, & \text{if } \mu_{\phi_V}(\text{i}) - k\sigma_{\phi_V}(\text{i}) > t_V \\ \perp, & \text{otherwise} \end{cases} \qquad \forall\ 1 \leq i \leq m_V$$

$$B_{SC}(i) = \begin{cases} 0, & \text{if } \mu_{\phi_{SC}}(\text{i}) + k\sigma_{\phi_{SC}}(\text{i}) < t_{SC} \\ 1, & \text{if } \mu_{\phi_{SC}}(\text{i}) - k\sigma_{\phi_{SC}}(\text{i}) > t_{SC} \\ \perp, & \text{otherwise} \end{cases} \quad \forall\ 1 \leq i \leq m_{SC}$$

for some threshold $t_V$ and $t_{SC}$ respectively (see Figure 3(j)). This phase is the training phase in our model. Here $k$ is a parameter to acquire a tradeoff between security and usability. With the increase in value of $k$, the user has better chance to generate the key successfully, but will hamper the security of the scheme. More precisely, the increase in the value of $k$ will increase the

false-positive rate and decrease the false-negative rate (as shown in our results in the evaluation Section 6).

The idea of defining the partial feature descriptor in this way is illustrated in Figure 5 (where the set $\{B, \mu, \sigma, t\}$ is replaced by $\{B_V, \mu_{\phi_V}, \sigma_{\phi_V}, t_V\}$ and $\{B_{SC}, \mu_{\phi_{SC}}, \sigma_{\phi_{SC}}, t_{SC}\}$ for voice and skin conductance respectively). If the $i^{th}$ feature descriptor is consistently same i.e. $\mu(i) + k\sigma(i) < t$ (the first case in Figure 5), then there is a high probability that the value of the $i^{th}$ feature descriptor will be less than $t$ during key reconstruction. Therefore, we can let the cell $T(i,0)$ of the lookup table contain a valid share of the key (and let $T(i,1)$ contain random bits). If the $i^{th}$ feature descriptor is consistently different, i.e. the value of the feature descriptor is unreliable (when compared to the threshold $t$ as in the third case in Figure 5), we let both $T(i,0)$ and $T(i,1)$ contain valid shares (typically different). Unlike [24], lookup tables are not encrypted (for discussion on this, see section 4.4).

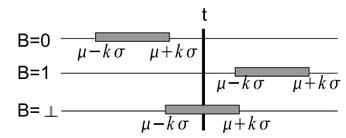



Figure 5: Definition of partial descriptor

Having valid shares in both $T(i,0)$ and $T(i,1)$ leads to different key shares used and consequently different keys being generated, which might not be desirable in systems that require a unique key. To solve this problem, a random cryptographic key $\mathcal{K}$ (unique for each user) is first generated, which is then encrypted with all possible valid keys ($K_{H_i}$) that can be derived from $<T_V \| T_{SC}>$. The key generation template therefore comprises of key $\mathcal{K}$ encrypted with $Z = |K_{H_i}|$ derived keys and the lookup tables $<T_V \| T_{SC}>$. Thus, the template = $<<T_V | T_{SC}>, <E_{K_{H_1}}(\mathcal{K} \| B), E_{K_{H_2}}(\mathcal{K} \| B), \ldots, E_{K_{H_Z}}(\mathcal{K} \| B)>>$, where $E_{\mathcal{K}_{H_i}}(msg)$ is a publicly known encryption algorithm and B is a unique string associated to each user which helps us to determine whether the decryption is correct or not in section 4.3.2.

#### 4.3.2 Cryptographic key reconstruction

When a user tries to reconstruct the cryptographic key, he/she first presents his/her spoken password and the skin conductance. The model collect this information, extracts the features and generates the feature descriptors for both voice and the SC. Corresponding shares from the lookup tables are chosen based on the feature descriptors.

$$b_V(i) = \begin{cases} 0 & \text{if } \phi_V(i) < t_V \\ 1 & \text{otherwise} \end{cases} \quad \forall \quad 1 \le i \le m_V$$

$$b_{SC}(i) = \begin{cases} 0 & \text{if } \phi_{SC}(i) < t_{SC} \\ 1 & \text{otherwise} \end{cases} \quad \forall \quad 1 \le i \le m_{SC}$$

For example, if the feature descriptor $\phi_{SC}(i)$ is less than the threshold $t_{SC}$, then $b_{SC}(i) = 0$ and $T_{SC}(i,0)$ is chosen from $T_{SC}$ as a key share; otherwise $b_{SC}(i) = 1$ and $T_{SC}(i,1)$ is chosen (see Figure 3(i)). $b_V$ and $b_{SC}$ are the feature keys and are obtained from voice and SC respectively.

A key $K'$ is derived by concatenating the key shares (see Figure 3(k)). This derived key is then used to decrypt the $|K_{H_i}|$ encrypted keys stored in the template. If the decryption succeeds (by matching the released B and the stored B), then the key $\mathcal{K}$ is released.

$$\mathcal{K}_D = \begin{cases} D_{K'}(E_{K_{H_i}}(\mathcal{K}|B)), & if \quad K' = K_{H_i} \\ Random, & if \quad K' \neq K_{H_i} \end{cases}$$

where, $D_{K'}(msg)$ is a publicly known decryption algorithm.

### 4.4 Discussions

While we try to use the consistency of voice and skin conductance to generate the correct key only when it is the genuine user in the normal emotional state, the inconsistency of voice and skin conductance poses challenges, too. Voice produced and skin conductance measured of the genuine user in a non-stressed emotional status might change due to tiredness, illness, noise, and etc.

We used an error correction technique, in particular, hamming distance, to improve the usability of the scheme. ${}^mC_d$ different keys are derived from any freshly generated key $K'$ obtained from the feature descriptors and $T$ (similar to the one derived in section 4.3.2), which are $d$ distance away from the derived key $K'$. All of these ${}^mC_d$ keys are then used to decrypt the encrypted keys before giving any negative answer to the user. If the decryption succeeds then the key $\mathcal{K}$ is released. For example, if $d = 2$ and length of the key is $m$, then ${}^mC_2$ different keys are derived. Thus, $|K_{H_i}| \times^m C_2$ decryptions are performed in attempting to recover $\mathcal{K}$.

Another issue concerns the privacy of the biometric data used. Ballard et al. propose using randomized biometric templates protected with low-entropy passwords to provide strong biometric privacy [4]. One can use this in conjunction with our model to provide both coercion resistance and biometric privacy. However, it is unclear whether the use of low-entropy passwords may have a negative impact on coercion resistance since, intuitively, an attacker may blame the user for providing the wrong low-entropy password in a coercion (similar problem discussed in section 3.2). We leave this as future work to develop a solution that satisfies both requirements.



## 5 Experimental Setup

We presented our design in generating a cryptographic key using voice and skin conductance in Section 4. It is important to test it out with real human beings to evaluate its performance. However, this is difficult as we need to find a way to make the participants feel stressed or nervous. It is clear that we cannot actually coerce them to do something by, e.g., putting a gun over their heads. Nevertheless, we performed case studies to induce stress on the participants and measure their voice and skin conductance. (IRB approval was obtained from our university before the user study.) We present the experimental setup in this section and the evaluation results and discussion in the next section.

### 5.1 Demographics

Since we were going to induce stress on the participants, we decided to concentrate on the younger generation (undergraduate and graduate students in the age from 18 to 30). We had altogether 43 participants, from which 4 participants detached the sensors from their fingers when they were nervous during the experiment. Therefore, we successfully performed our experiments on 39 participants, out of which 22 were male and 17 were female.

### 5.2 Experimental settings

Participants were asked to sit in a small office where the overhead fluorescent lights were turned off and a dim red incandescent lamp was turned on to reduce the possible electrical interference with the monitoring equipments. The room was air conditioned to approximately 72°F and humidity level was generally dry. This is done in accordance to the variation of skin conductance in different environmental conditions [36].

Skin conductance sensors[1] were attached to the three middle fingers of the participant to record SC (shown in Figure 2). The participant was also asked to keep her left hand (with sensors attached) as still as possible to avoid interference from the sensors. Fake heart rate tags were tied to the wrist, which gave an illusion of monitoring the heart rate.

Initially, there was an incomplete disclosure regarding the purpose and the steps of the study in order to ensure that the participant's responses will not be affected by her knowledge of the research.

### 5.3 Procedure

We ran two experiments (`e1` and `e2`). Each experiment consisted of two parts, where the first parts (`e1n` and `e2n`) were conducted when the participants were in a normal (calm) condition, and the second parts (`e1s` and `e2s`) were conducted when the participants were stressed.

We ran experiment `e1n` by

- showing nice (geographical) pictures one after another and short phrases (the spoken password embedded) which are related to the pictures, and asking the participant to read them out;
- showing fake visual heartbeats at a normal rate at the bottom of the screen and correspondingly playing heartbeats sound.

In order to capture the emotional responses in the stress scenario in `e1s`,

- a frightening horror movie was played, replacing the nice pictures;
- the rate of the heartbeats were gradually increased to induce more stress on the participant;
- the participant was asked to read out some short phrases at the end of each horror scene (rather than along with the video) to avoid distraction.

Similar studies have been performed previously to measure the stress level in users [26, 19].

In `e2`, we went a bit further to induce more stress on the participant. Figure 6 shows the change in skin conductance in response to different events in `e2`. During `e2`, the participant was asked to type a few sentences (e.g., "Work is much more fun than fun") shown to her in a fixed period of time. She was also warned (prior to the experiment) not to press the "ALT" key on the keyboard, as it would cause the computer program to crash and all data would be lost (event A). We then left the participant alone in the room to continue typing (event B). We configured the computer to restart after 3 minutes irrespective of whether the participant actually touched the "ALT" key or not. The computer would then boot from a USB drive into MS-DOS and display some error messages (event C). This completes the first part of `e2`, i.e., `e2n`.

Stress started to develop at this point in time as the participant believed that she had pressed the "ALT" key which caused data loss on the computer (event D). We purposely left the participant alone so that stress could develop further and she could not get immediate help to resolve the "problem". After that, the researcher entered the room and examined the keyboard and the computer (event E) and then accused the participant of her negligent act of pressing the "ALT" key (event F). This turned

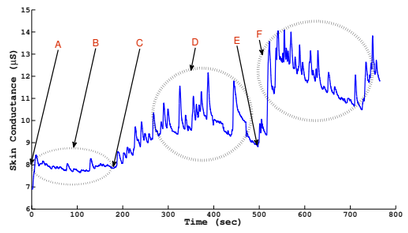



Figure 6: Change of skin conductance in e2

out to be successful in making the participant stressed as we observed that many participants were nervous at this point in time. Some kept saying "sorry"; some tried very hard to fix the "problem", and some started calling for help. There were also voluntary confession statements from the participants, e.g., "I hit the ALT key by mistake in place of typing the 'X' key", "It was a mistake from my side.".

### 5.4 Discussion

In this section, we discuss the difference of the emotional state of a user in real life and in our user study, and limitations of our experiment.

1. Training of the system
   - Real life: the user is in a (controlled) environment specified by our system, in which the stress level is low. This allows us to generate the lookup table for that particular user with the normal skin conductance level.
   - User study: the user is in exactly the (controlled) environment specified by our system, i.e., when watching a relaxation movie.
2. Trying to generate the cryptographic key; no coercion
   - Real life: a user could be in various emotional states, including being happy, sad, angry, etc.
   - User study: same as in training when the user is watching a relaxation movie. In this work, we only try to analyze how our system performs when users are calm and relaxed. It remains future work to analyze how it works when the user is in other emotional states. We do expect the false-negative rate to rise when the user is in other emotional states.
3. Trying to generate the cryptographic key; in coercion
   - Real life: a user can be forced/coerced in many different ways, e.g., a gun to the head, or a knife under the throat, etc.
   - User study: watching a horror movie and being forced to plead guilty (having damaged a notebook computer). We tried our best to approximate the real-life scenarios, but there is a limit we could go when doing this to real human beings (e.g., IRB restriction). However, we believe that what we did is a clever way of studying human behavior when being coerced.

Discussions above highlight some limitations of our scheme, e.g., we have not tested how it reacts to other emotional status (happy, sad, angry, etc.) and how skin conductance may change naturally (due to oily fingers, etc.). There are two other important limitations in the present study. First, our study does not test the repeatability of using our scheme, i.e., we did not ask the participants to come back and try again. The second limitation comes with the over-controlled environment, e.g., quiet office (because of the use of voice), controlled temperature and humidity [9](because of the use of skin conductance), and etc. It remains further work to test our scheme in different settings.

## 6 Evaluation and Discussion

In this section, we analyze the data collected in our user study. We first describe how we partition the data into different groups (e.g., for training and test purposes), see Section 6.1. We then present a series of analysis on the false-positive and false-negative rates (Section 6.2). Finally we show the change in the password space where an attacker has perfect knowledge of our design and the content stored. We show that this change in the password space in this worst case is small (Section 6.3).

### 6.1 Training and Testing Datasets

We have collected voice and skin conductance signals for 39 participants. For each participant, we have collected many samples of the signals when the participant is either calm or stressed. Table 2 shows the number of samples we collected in each experiment for each participant. Voice signals are typically 2 to 3 seconds long, while skin conductance signals are about 10 seconds long to avoid fluctuations.

Figure 7 shows how we obtain dataset to

- split original sample sets $\{\nu^{\mathsf{full}}_{\mathsf{e1}n}, \omega^{\mathsf{full}}_{\mathsf{e1}n}, \omega^{\mathsf{full}}_{\mathsf{e2}n}\}$ into two equal halves $\{\nu^{\mathsf{train}}_{\mathsf{e1}n}, \omega^{\mathsf{train}}_{\mathsf{e1}n}, \omega^{\mathsf{train}}_{\mathsf{e2}n}\}$ and $\{\nu^{\mathsf{test}}_{\mathsf{e1}n}, \omega^{\mathsf{test}}_{\mathsf{e1}n}, \omega^{\mathsf{test}}_{\mathsf{e2}n}\}$ to obtain datasets for training and testing (see the half circles);



| Feature | | e1n | e1s | e2n | e2s |
|---|---|---|---|---|---|
| Voice | # of samples | 26 | 5 | 0 | 0 |
| | Notation | $\nu^{\mathsf{full}}_{\mathsf{e1}n}$ | $\nu^{\mathsf{full}}_{\mathsf{e1}s}$ | - | - |
| SC | # of samples | 26 | 60 | 18 | 60-80 |
| | Notation | $\omega^{\mathsf{full}}_{\mathsf{e1}n}$ | $\omega^{\mathsf{full}}_{\mathsf{e1}s}$ | $\omega^{\mathsf{full}}_{\mathsf{e2}n}$ | $\omega^{\mathsf{full}}_{\mathsf{e2}s}$ |

Table 2: Number of samples collected for each participant

- combine different voice samples and skin conductance samples to create new datasets to test our system (see circles in the middle column). $\{\nu^{\mathsf{train}}_{\mathsf{e1}n}$ & $\omega^{\mathsf{train}}_{\mathsf{e1}n}\}$, $\{\nu^{\mathsf{test}}_{\mathsf{e1}n}$ & $\omega^{\mathsf{test}}_{\mathsf{e1}n}\}$, $\{\nu^{\mathsf{train}}_{\mathsf{e1}n}$ & $\omega^{\mathsf{train}}_{\mathsf{e2}n}\}$, $\{\nu^{\mathsf{test}}_{\mathsf{e1}n}$ & $\omega^{\mathsf{test}}_{\mathsf{e2}n}\}$ are combined to create $\{\xi^{\mathsf{train}}_{\mathsf{e1}n}\}$, $\{\xi^{\mathsf{test}}_{\mathsf{e1}n}\}$, $\{\xi^{\mathsf{train}}_{\mathsf{e2}n}\}$, $\{\xi^{\mathsf{test}}_{\mathsf{e2}n}\}$ respectively.
- to obtain the stress dataset $\{\nu^{\mathsf{full}}_{\mathsf{e1}s}$ & $\omega^{\mathsf{full}}_{\mathsf{e1}s}\}$, $\{\nu^{\mathsf{full}}_{\mathsf{e1}s}$ & $\omega^{\mathsf{full}}_{\mathsf{e2}s}\}$ are combined to create $\{\xi^{\mathsf{full}}_{\mathsf{e1}s}\}$, $\{\xi^{\mathsf{full}}_{\mathsf{e2}s}\}$ respectively.

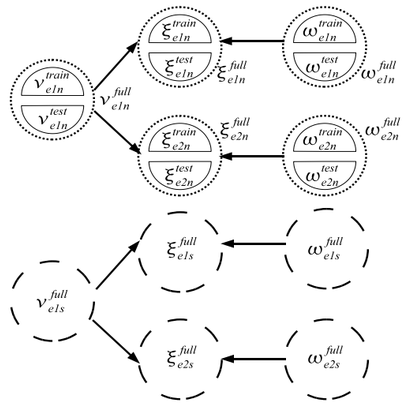



Figure 7: Splitting and combining datasets

Note that the voice and skin conductance samples that are combined together might not have been captured at exactly the same time. We allow a time gap because an attacker might record the voice of the victim to be used in conjunction with the skin conductance of the victim at a slightly different time. Both samples were captured in the same part of the experiment, though, i.e., both from e1s or both from e2s.

### 6.2 Accuracy of our model

The false-negative rate of our system is defined as the percentage of failed login attempts by a legitimate user with her cryptographic key generated, averaged over all users in a population $A$. Similarly, the false-positive rate is defined as the percentage of failed detection of attempts by illegitimate users or legitimate users in a stressful situation, averaged over all users in a population $A$.

**Voice samples only** We first evaluate the voice samples we collected in our experiments. The purpose is to check out the false-positive and false-negative rates, in an event if only voice samples are used to generate cryptographic keys. The system is trained with $\nu^{\mathsf{train}}_{\mathsf{e1n}}$ of user $a_i$, and is tested against $\nu^{\mathsf{full}}_{\mathsf{e1n}}$ of user $a_j$ where $i \neq j$, $\forall\ j \in A$ to calculate the false-positive rates; and against $\nu^{\mathsf{test}}_{\mathsf{e1n}}$ of user $a_i$ to calculate the false-negative rates. Results are averaged on all users in $A$. We try different random $\alpha_V$ vectors and choose the one that yields the smallest sum of the false-positive and false-negative rates. We try different settings of the hamming distance parameter $d$, and find that 2 gives a reasonable tradeoff between false-positive and false-negative rates. The false-positive and false-negative rates for different values of $k$ are plotted in Figure 8.

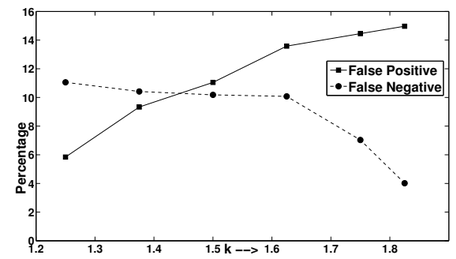
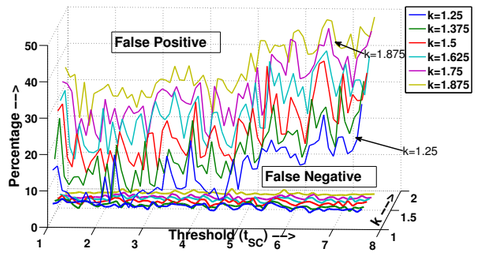


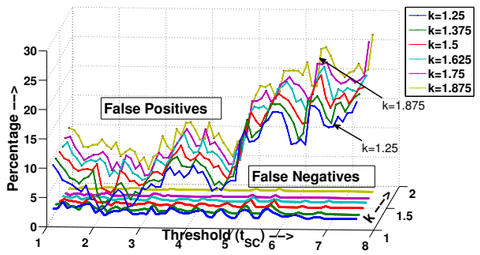

Figure 8: False-positive and false-negative rates for spoken passwords

Figure 8 shows that we manage to get a comparable accuracy with the previous work [24] in terms of the false-negative rate. False-positive rate was not reported in [24].

**Skin conductance only** Next, we evaluate the skin conductance samples to see how well they reflect the change in the participants' emotional status. We show the results in Figure 9(a) and Figure 9(b) for experiment e1 and e2, respectively. The different color lines denotes different 'k' values in Figure 9 and Figure 10. The system is trained with $\omega^{\mathsf{train}}_{\mathsf{e1n}}$ (and $\omega^{\mathsf{train}}_{\mathsf{e2n}}$, respectively) of user $a_i$, and is tested against the stressed full data set, $\omega^{\mathsf{full}}_{\mathsf{e1s}}$ (and $\omega^{\mathsf{full}}_{\mathsf{e2s}}$, respectively) of the same user $a_i$ to calculate the false-positive rates; or against the normal test data set, $\omega^{\mathsf{test}}_{\mathsf{e1n}}$ (and $\omega^{\mathsf{test}}_{\mathsf{e2n}}$, respectively) of the same user $a_i$ to calculate the false-negative rates. Results are averaged over all users in $A$.




(a) e1




(b) e2

Figure 9: False-positive and false-negative rates for skin conductance

Note that the false-positive and false-negative rates are higher for e1 in Figure 9(a). We believe, this is because of the reason that the intensity of some of the horror videos was not very high, which did not result in a noticeable change in the skin conductance for many users.

We can observe the tradeoff of various settings of $k$ and the threshold from these figures. In general, this shows that whenever a user is under stress, her skin conductance can be used to differentiate between the two emotional state with good accuracy. For example in e2, when $k = 1.25$ and $t_{SC} = 2.1$, we obtained a false-positive rate of $3.2\%$ and a false-negative rate of $2.2\%$ (see Figure 9(b)). If we increase the value of $k$ from 1.25 to 1.875 in both Figures 9(a) and 9(b), we could see a decrease in the false-negative rates (increasing usability) and increase in the false-positive rates (compromising with the security). We used the hamming distance parameter $d = 2$ in our setting.

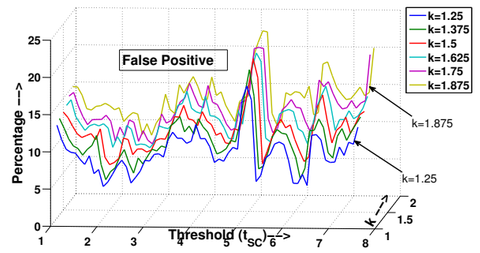
**Voice combined with skin conductance** Voice and skin conductance samples are combined as shown in Figure 7 to obtain the samples needed in this evaluation. We first train the system with $\xi_{\text{e2n}}^{\text{train}}$, and then evaluate the system against three different datasets to evaluate the false-positive and false-negative rates.

a $\xi_{\text{e2n}}^{\text{full}}$ of user $a_j$ where $i \neq j, \forall\, j \in A$: when a different person tries to generate the key (Figure 10(a));

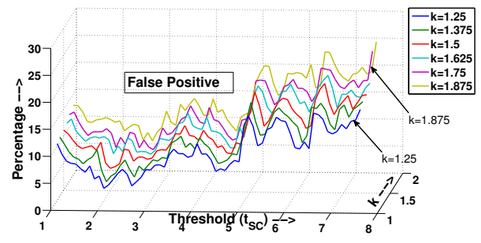
b $\xi_{\text{e2s}}^{\text{full}}$ of user $a_i$: when the same user tries to generate the key when she is being coerced (Figure 10(b));

c $\xi_{\text{e2n}}^{\text{test}}$ of user $a_i$: when the same user tries to generate the key when she is not being coerced (Figure 10(c)).

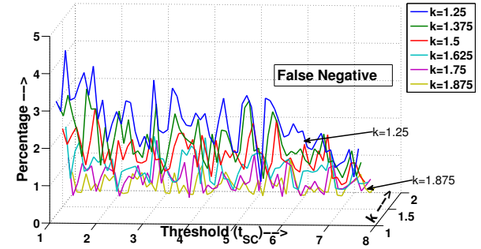
We evaluate the false-positive rates in the first two cases and the false-negative rates in the third case. Results are averaged over all users in $A$. We use a hamming distance parameter $d = 4$, and show the results in Figure 10.




(a) False-positive against $\xi_{\text{e2n}}^{\text{full}}$ of user $a_j$ $i \neq j$




(b) False-positive against $\xi_{\text{e2s}}^{\text{full}}$ of user $a_i$




(c) False-negative against $\xi_{\text{e2n}}^{\text{test}}$ of user $a_i$

Figure 10: False-positive and false-negative rates for voice combined with skin conductance



These results show that generating cryptographic keys from voice and skin conductance is effective in fighting coercion attacks, as we observe false-positive rates between $6\%$ to $15\%$ for $1 \leq t_{SC} \leq 4$, which can also rise up to $22\%$ for $t_{SC} \geq 5$. False-negative rates are between $0\%$ and $4.5\%$ for all values of $t_{SC}$. Further efforts are needed to reduce the false-positive and false-negative rates. Same as in the previous subsection, if we increase the value of $k$ from 1.25 to 1.875, we could see a decrease in the false-negative rates and increase in the false-positive rates.

## 6.3 Change in password space

In this subsection, we discuss more advanced attacks on our system (if implemented) beside forcing the victim to obtain her spoken password and skin conductance. If such system is implemented, then we need to approximate the entropy in the worst case of these advanced attacks, in which the attacker makes use of the group information about the skin conductance and information stored in the key generation module.

The group information about skin conductance refers to the patterns observed in the change in the users' feature key generated from the skin conductance ($b_{SC}$) when they are coerced. An attacker could use this information to selectively modify the victims skin conductance feature key in order to improve the probability of generating the correct key. To know how we obtained the feature key ($b_{SC}$) for SC, see section 4.

Although we do not store any biometric information of the user directly on the device (see discussions in Section 4), we still need to store the lookup tables ($T_V$ and $T_{SC}$) which are derived from the user specific data (e.g., feature descriptors). Although this table can be encrypted with a user password as discussed in previous work [24], however we try not to rely the security of our model on the secrecy of this table because we are dealing with coercion attacks. In the rest of this subsection, we assume that an attacker has perfect knowledge in both the group information about skin conductance and the lookup tables. We want to approximate the guessing entropy, i.e., the reduction in the password space for this more powerful attacker.

More precisely, we assume in the worst case that an attacker has access to

- the lookup tables $T_V$ and $T_{SC}$;
- the recorded spoken password of the user and the corresponding feature key $\{b_V(i)\}$;
- the recorded skin conductance when the user is stressed and the corresponding feature key $\{b_{SC}^S(i)\}$;
- the database $D$ which contains the mapping of the SC feature keys when users are normal ($\{b_{SC}^N(i)\}$) to the scenario when they are stressed ($\{b_{SC}^S(i)\}$) for all users in a population $A$.

A sample database $D$ for such mapping of SC is shown in Table 3 for $|A|$ users. Each row in the table is a record of the feature key of a user when she is normal and stressed, and the last column shows the index of the feature keys that had changed from $b_{SC}^N$ to $b_{SC}^S$.

| # | $b_{SC}^N$ | $b_{SC}^S$ | Flipped bits' pos. |
|---|---|---|---|
| 1 | 011011011011 | 001101110011 | 2,4,5,7,9 |
| 2 | 010010010111 | 010100110110 | 4,5,7,12 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| $\|A\|$ | 010101001100 | 111111100110 | 1,3,5,7,9,11 |

Table 3: A sample Database $D$

The attacker's strategy would be to analyze $D$ to learn patterns in which people's feature keys $\{b_{SC}^N\}$ changes to $\{b_{SC}^S\}$, e.g., whenever the $i$-th index of the feature key changes, the $j$-th one will change too.

These patterns can be easily learned by applying a well studied technique called association rule mining [2]. The attacker can then use these patterns to reduce the password space. Here, we use a simple example to demonstrate the idea.

We first represent the password space by a sequence of 0's (the corresponding index in $\{b_{SC}^S\}$ will definitely not change when a user's emotional status changes), 1's (the corresponding index in $\{b_{SC}^S\}$ will definitely change), and $*$'s (don't know), e.g., $[1, *, *]$ represents a password space in which only the first index of $\{b_{SC}^S\}$ will change, and therefore the password space is $2^2 = 4$. When the attacker makes use of a pattern learned, e.g., "the change of the first index of $\{b_{SC}^S\}$ implies the change of the second one", he can convert the password space from $[1, *, *]$ to $[1, 1, *]$, since the second index of the $\{b_{SC}^S\}$ will definitely change, too. With this, the password space reduces to $2^1 = 2$.

We present the detailed algorithm with an example in estimating this reduction in the password space in the Appendix A.

We constructed the database $D$ with the skin conductance samples collected in our user study, mine all association rules, and then use the above algorithm to find out the change in the password space. Figure 11 shows the results for different settings of the threshold and minimum confidence in the association rule mining.
$k$ is set to 1.25 in this experiment, and the minimum support is set to $30\%$. Note that the original password space is $2^{m_{SC}} = 2^{50}$. Although in the worst case the effective number of bits to represent the password space reduces

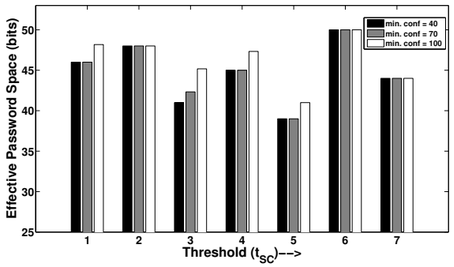



Figure 11: Password Space reduction

by roughly $20\%$, many settings of the threshold value result in only $10\%$ reduction.

Another way to attack our system is to make the user take a sedative to relieve his/her anxiety before capturing SC. The attacker can then use this skin conductance to generate the key. We are trying to collaborate with medical practitioners and researchers to see the correlation between the two skin conductances, one under normal condition without taking any sedative and the other under coercion and having taken the sedative. For now this remains as a future work.

## 7 Conclusion and Future Work

In this paper we present a novel approach for fighting against coercion attacks in generating cryptographic keys using *skin conductance* (SC) of a person. In coercion attack, the attacker forces a user to grant him access to the system. SC was used to determine the person's overall arousal state i.e. (emotional status). The change in the emotional status of a person results in different keys. We discussed the reasons of adopting SC as an emotional response parameter and why it was preferred over other physiological signals like Electrocardiography, Electromyography, Heart Rate, respiration, skin temperature etc. In this paper, we have chosen skin conductance along with voice in generating cryptographic keys; however, one can choose any other biometric for e.g. iris, fingerprint, face etc. in lieu of voice. Cryptographic key is generated using lookup table method as discussed in [24].

In our knowledge the presented work is the first in fighting coercion attacks in generating cryptographic keys. We conducted two experiments in our user study and have shown some interesting results. The proposed model was tested with 39 user's voice and skin conductance data to compute the false-positive and false-negative rate. Furthermore our results showed that the cryptographic key generated in two different scenarios are different for the same person. This bolsters our heuristic to use skin conductance for fighting against coercion attacks. As both skin conductance and voice are not static biometrics, in some cases we obtained high false-negatives. We evaluated the security of the proposed model in terms of entropy and several threat models and discussed how difficult it is for an attacker, in an event when she has full information about the key generation module; the skin conductance of the victim in the stressful scenario; and the group information about the skin conductance.

Note that guessing entropy and guessing distance [5] might provide deeper insight in the security of our model. We leave it as our future work. In terms of feasibility, in future we will also like to see in some possibilities of building the system (may be a mobile device) with all three: voice, skin conductance and fingerprint extraction mechanism to authenticate to the system. Furthermore, we would like to look into other emotional responses like happy, joy, anger, sad etc., to make the claim of using SC in fighting coercion attacks stronger. This paper does not study the repeatability of the key using the proposed scheme and is left as a future work.

**Acknowledgments** We are deeply grateful to Lucas Ballard for comments and suggestions in key construction, as well as David Lo for helpful discussions on the analysis of the password space.

## References

[1] ABHYANKAR, A., AND SCHUCKERS, S. Integrating a wavelet based perspiration liveness check with fingerprint recognition. *Pattern Recognition 42*, 3 (2009), 452–464.

[2] AGRAWAL, R., IMIELIŃSKI, T., AND SWAMI, A. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1993), ACM, pp. 207–216.

[3] ANTTONEN, J., AND SURAKKA, V. Emotions and heart rate while sitting on a chair. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 2005), ACM, pp. 491–499.

[4] BALLARD, L., KAMARA, S., MONROSE, F., AND REITER, M. K. Towards practical biometric key generation with randomized biometric templates. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), ACM, pp. 235–244.

[5] BALLARD, L., KAMARA, S., AND REITER, M. K. The practical subtleties of biometric key generation. In *SS'08: Proceedings of the 17th conference on Security symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 61–74.

[6] BIEL, L., PETTERSSON, O., PHILIPSON, L., AND WIDE, P. Ecg analysis: a new approach in human identification. *Instrumentation and Measurement, IEEE Transactions on 50*, 3 (Jun 2001), 808–812.

[7] CACIOPPO, J. T., AND TASSINARY, L. G. Inferring psychological significance from physiological signals. *American Psychologist 45*, 1 (Jan 1990), 16–28.

[8] CAMPBELL, J. P. Speaker recognition: a tutorial. *Proceedings of the IEEE 85*, 9 (1997), 1437–1462.

[9] CONKLIN, J. E. Three Factors Affecting the General Level of Electrical Skin-Resistance. *The American Journal of Psychology 64*, 1 (Jan 1951), 78–86.

[10] DAVIS, S., AND MERMELSTEIN, P. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on 28*, 4 (1980), 357–366.

[11] EKMAN, P. *Basic Emotions*, vol. 476 of *Handbook of Cognition and Emotion*. T. Dalgleish and M. Power, John Wiley & Sons Ltd. Sussex UK, 1999.

[12] ENCOSKIN. ENCOLL's Collagen Technology, 2010. http://www.encoll.com/SkinCare_and_Dietary_Products.htm.

[13] FANG, Z., GUOLIANG, Z., AND ZHANJIANG, S. Comparison of different implementations of mfcc. *J. Comput. Sci. Technol. 16*, 6 (2001), 582–589.

[14] FOWLES, D. C. The Three Arousal Model: Implications of Gray's Two-Factor Learning Theory for Heart Rate, Electrodermal Activity, and Psychopathy. *Psychophysiology 17*, 2 (1980), 87–104.

[15] GARCÍA PERERA, L., NOLAZCO FLORES, J., AND MEX PERERA, C. Cryptographic-Speech-Key Generation Architecture Improvements. In *IbPRIA05* (2005), p. II:579.

[16] HELANDER, M. Applicability of drivers' electrodermal response to the design of the traffic environment. *Journal of Applied Psychology 63*, 4 (1978), 481–488.

[17] INVESTIGATION, F. B. O. Bank crime statistics (bcs) federal insured financial institutions july 1, 2009 september 30, 2009. http://www.fbi.gov/publications/bcs/bcs2009/bank_crime_2009q3.htm.

[18] JAIN, A. K., ROSS, A., AND PANKANTI, S. Biometrics: a tool for information security. *IEEE Transactions on Information Forensics and Security 1*, 2 (2006), 125–143.

[19] JOHNSON, K. J., AND FREDRICKSON, B. L. We all look the same to me: Positive emotions eliminate the own-race bias in face recognition. *Psychological Science 16* (2005), 875–881.

[20] KIM, J. Bimodal Emotion Recognition using Speech and Physiological Changes. In *In M. Grimm, K. Kroschel (Ed.), Robust Speech Recognition and Understanding*. I-Tech Education and Publishing, Vienna, Austria, 2007, pp. 265–280.

[21] KIM1, K. H., BANG, S. W., AND KIM, S. R. Emotion recognition system using short-term monitoring of physiological signals. *Medical and Biological Engineering and Computing 42*, 3 (May 2004), 419–427.

[22] LAPSLEY, P. D., LEE, J. A., PARE, JR., D. F., AND HOFFMAN, N. Anti-fraud biometric scanner that accurately detects blood flow. US Patent # 5737439, 1998.

[23] LEE, C. K., YOO, S. K., PARK, Y., KIM, N., JEONG, K., AND LEE, B. Using Neural Network to Recognize Human Emotions from Heart Rate Variability and Skin Resistance. In *27th Annual International Conference of the Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005* (2005), pp. 5523–5525.

[24] MONROSE, F., REITER, M. K., LI, Q., AND WETZEL, S. Cryptographic Key Generation from Voice(Extended Abstract). In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2001), IEEE Computer Society, p. 202.

[25] MONROSE, F., REITER, M. K., AND WETZEL, S. Password hardening based on keystroke dynamics. *International Journal of Information Security 1*, 2 (2002), 69–83.

[26] NACHSON, I., AND FELDMAN, B. Psychological Stress Evaluator - Validity Study. *Crime and Social Deviance 7*, 2 (1979), 65–81.

[27] NANDAKUMAR, K., JAIN, A. K., AND PANKANTI, S. Fingerprint-Based Fuzzy Vault: Implementation and Performance. *IEEE Transactions on Information Forensics and Security 2*, 4 (2007), 744–757.

[28] NANDAKUMAR, K., NAGAR, A., AND JAIN, A. Hardening Fingerprint Fuzzy Vault Using Password. In *ICB07* (2007), Springer Berlin / Heidelberg, pp. 927–937.

[29] NASOZ, F., ALVAREZ, K., LISETTI, L., AND FINKELSTEIN, N. Emotion recognition from physiological signals using wireless sensors for presence technologies. *Cognition, Technology and Work 6*, 1 (2004), 4–14.

[30] PHUA, K., CHEN, J., DAT, T. H., AND SHUE, L. Heart sound as a biometric. *Pattern Recogn. 41*, 3 (2008), 906–919.

[31] PICARD, R. W., VYZAS, E., AND HEALEY, J. Toward Machine Emotional Intelligence: Analysis of Affective Physiological State. *IEEE Transaction Pattern Analysis Matching Intelligence 23*, 10 (2001), 1175–1191.

[32] PRABHAKAR, S., PANKANTI, S., AND JAIN, A. K. Biometric recognition: Security and privacy concerns. *IEEE Security and Privacy 1*, 2 (2003), 33–42.

[33] RATHA, N. K., CONNELL, J. H., AND BOLLE, R. M. An Analysis of Minutiae Matching Strength. In *AVBPA '01: Proceedings of the Third International Conference on Audio- and Video-Based Biometric Person Authentication* (London, UK, 2001), Springer-Verlag, pp. 223–228.

[34] SAKAI, K., AND QUICK, T. W. Moisturizing skin preparation, July 1988.

[35] SANTOS A, M. F., AGUILAR A, J. F., AND GARCIA A, J. O. Cryptographic key generation using handwritten signature. *Biometric Technologies for Human Identification III, Processing of SPIE* (2006).

[36] SCHMIDT, S., AND WALACH, H. Electrodermal Activity (EDA) - State of the Art Measurement and Techniques for Parapsychological Purposes. *Journal of Parapsychology 64* (June 2000), 139 – 163.

[37] SELYE, H. *The Stress of Life*. McGraw-Hill, 1956, ch. 1-7.

[38] TO THE WILD DIVINE, J. *Skin conductance aquisition device, Lightstone*. http://www.wilddivine.com/.

[39] UIDIA. Unique Identification Authority of India Card Project-India. http://www.uidaicards.com/.

[40] ULUDAG, U., AND JAIN, A. Attacks on biometric systems: a case study in fingerprints. In *Proc. SPIE-EI 2004, Security, Seganography and Watermarking of Multimedia Contents VI* (2004), pp. 622 – 633.

[41] VALENTINE, D. Skin Conductance One Of The Fastest Ways To Test Stress, 2009. http://www.articlesbase.com/health-articles/skin-conductance-one-of-the-fastest-ways-to-test-stress-1464442.html, [Online; accessed 16-November-2009].

[42] WESTERINK, J. H. D. M., VAN DEN BROEK, E. L., SCHUT, M. H., VAN HERK, J., AND TUINENBREIJE, K. *Computing Emotion Awareness Through Galvanic Skin Response and Facial Electromyography*, vol. 8 of *Philips Research Book Series*. Springer Netherlands, New York, December 2007.

## A Guessing Entropy for Skin Conductance

Let $R$ be the set of rules the attacker can use to reduce the password space from $\mathsf{S}$ to $\mathsf{S}'$. So, for a rule $R_i$

$$antecedent(\mathsf{A}) \Rightarrow consequent(\mathsf{C})$$

such that, A=[$y_1, \ldots, y_{E_a}$] and C=[$z_1, \ldots, z_{E_c}$], where $E_a$ are the elements in the antecedent and $E_c$ in consequent. The process of calculating the new password space from a given one is shown in algorithm 2. S′ indicates a lower bound for the password space which shows the minimum number of combinations an attacker needs to guess if he has a full knowledge of the mappings in the database.

Let $\Psi$ denote the candidate set and $\Phi$ be the large itemset, $\Psi^I$ and $\Phi^I$ are the two dimensional vectors derived from the rules $R_1, \ldots, R_I$. Each item ($\Psi^I_J$) in a $\Psi^I$ is a vector of the form $[x_1, x_2, \ldots, x_{m_{SC}}]$, $\forall\ 0 \leq J \leq L$, where $x_i \in (0, 1, *)$ and $L = |\Psi^I|$. Similarly, each item ($\Phi^I_J$) in a $\Phi^I$ is also vector of the form $[x_1, x_2, \ldots, x_{m_{SC}}]$, $\forall\ 0 \leq J \leq L$, where $x_i \in (0, 1, *)$ and $L = |\Phi^I|$.

**Algorithm 2** Reduced Password Space for SC

PasswdSpace $(R)$

1: $\Phi^0_1 \longleftarrow [*, *, *, *, *, *, *, *, \ldots, *]$
2: S $\longleftarrow 2^{m_{SC}}$
3: **for** I = 1 to $|R|$ **do**
4: L $\longleftarrow \mathbf{length}(\Phi^{I-1})$
5: $\Psi^I \longleftarrow$ NULL
6: **for** J = 1 to L **do**
7: **if any** $\left((\Phi^{I-1}_{J,y_1}, \Phi^{I-1}_{J,y_2}, \ldots, \Phi^{I-1}_{J,y_{E_a}}) == *\right)$ **then**
8: $\Psi^I \longleftarrow \Psi^I \bigcup \mathbf{split}(\Phi^{I-1}_J)$
9: **else**
10: $\Psi^I \longleftarrow \Psi^I \bigcup \Phi^{I-1}$
11: **end if**
12: **end for**
13: $\Psi^I \longleftarrow \mathbf{unique}\left(\Psi^I\right)$
14: cnt $\longleftarrow$ 1
15: L $\longleftarrow \mathbf{length}(\Psi^I)$
16: **for** J = 1 to L **do**
17: **if** $\left(\prod_{p=1}^{E_a} \Psi^I_{J,y_p} == 1 \& \prod_{q=1}^{E_c} \Psi^I_{J,z_q} == 0\right)$ **then**
18: $\mathbf{delete}\left(\Psi^I_J\right)$
19: **else**
20: $\Phi^I_{cnt} \longleftarrow \Psi^I_J$
21: cnt++
22: **end if**
23: **end for**
24: **end for**
25: S′ $\longleftarrow \Phi^{|R|}$

$*$ denotes *don't care* and can be assigned 0 or 1. The set of rules $R$ obtained are passed to the algorithm 2 to generate S′. $\Phi^0_1$ is initialized to $[* * * * * * * \ldots *]$ and S = $2^{m_{SC}}$. Below is the short description of the functions used in the algorithm.

- $\mathbf{length}(\Psi^I)$ - gives the total vectors in the candidate set $\Psi^I$ i.e. $|\Psi^I|$.
- $\mathbf{any}(\Phi^J_I(y_1, y_2, \ldots, y_{E_a}) == *)$ - a boolean function
$$= \begin{cases} \mathbf{1}, & (\Phi^I_{J,y_1} == *) \vee \ldots \vee (\Phi^I_{J,y_{E_a}}) == *) \\ \mathbf{0}, & \text{else} \end{cases}$$
- $\mathbf{split}(\Phi^I_J)$ - this function generates a new candidate set $\Psi^I$ from a large itemset $\Phi^{I-1}$ based on a rule $R_I$. It generates the vectors for $\Psi^I$ *s.t.*
  - Mark $y_i$, if $(\Phi^{I-1}_{J,y_i} == *)$, $\forall\ y_i \in [y_1, \ldots, y_{E_a}]$.
  - Generate all possible combination of the marked bits; which implies if total number of marked bits are $mb$ then total possible combinations are $2^{mb}$. For e.g. if $\Phi^I_J$ = [***1*1] and the rule $R_I$ is $1 \Rightarrow 2$, then the result is $([11 * 1 * 1]\ [10 * 1 * 1]\ [01 * 1 * 1]\ [00 * 1 * 1])$
- $\mathbf{unique}(\Psi^I)$ - gives the unique vectors from $\Psi^I$.
- $\mathbf{delete}(\Psi^I_J)$ - delete $\Psi^I_J$ from the candidate set $\Psi^I$.

During the Candidate Itemset Generation, a $*$ in the large itemset triggers a *split*; 1 and 0 indicates *do nothing*. However during the Large Itemset Generation a 1 in a candidate itemset triggers *add 1*; 0 indicates *do nothing*. During the whole procedure, each time one rule is used and the sets which does not comply with that rule are omitted to create the new set. The final password space is calculated by computing the total number of vectors which can be generated using $\Phi^{|R|}$, where $\Phi^{|R|}$ is the final large itemset generated from the rules $R_1, \ldots, R_{|R|}$.

An example shown in Table 4 with *5* elements, to how to generate the candidate itemset and the large itemset from *3* rules. The total number of guesses which an attacker needs to make is *14* which implies the effective number of bits in the new password space are *4*; original was *5*.

| $R$ | | Candidate Itemset | | Large Itemset |
|---|---|---|---|---|
| Initialization | | | $\Phi^0_1$ | $* * * * *$ |
| $\mathbf{R_1}$ $1 \Rightarrow 3$ | $\Psi^1_1$ | $1 * * * *$ | $\Phi^1_1$ | $1 * 1 * *$ |
| | $\Psi^1_2$ | $0 * * * *$ | $\Phi^1_2$ | $0 * * * *$ |
| $\mathbf{R_2}$ $(1, 2) \Rightarrow 5$ | $\Psi^2_1$ | $101 * *$ | $\Phi^2_1$ | $101 * *$ |
| | $\Psi^2_2$ | $111 * *$ | $\Phi^2_2$ | $111 * 1$ |
| | $\Psi^2_3$ | $01 * **$ | $\Phi^2_3$ | $01 * **$ |
| | $\Psi^2_4$ | $00 * **$ | $\Phi^2_4$ | $00 * **$ |
| $\mathbf{R_3}$ $5 \Rightarrow 1$ | $\Psi^3_1$ | $101 * 0$ | $\Phi^3_1$ | $101 * 0$ |
| | $\Psi^3_2$ | $101 * 1$ | $\Phi^3_2$ | $101 * 1$ |
| | $\Psi^3_3$ | $111 * 1$ | $\Phi^3_3$ | $111 * 1$ |
| | $\Psi^3_4$ | $01 * *1$ | $\Phi^3_4$ | $01 * *0$ |
| | $\Psi^3_5$ | $01 * *0$ | $\Phi^3_5$ | $00 * *0$ |
| | $\Psi^3_6$ | $00 * *1$ | | |
| | $\Psi^3_7$ | $00 * *0$ | | |

Table 4: Generating candidate set and large itemset

## Notes

[1] We use a physiological data acquisition device called Lightstone from WildDivine [38].