Mugshot: Deterministic Capture and Replay for JavaScript Applications

James Mickens, Jeremy Elson, and Jon Howell
Microsoft Research
mickens,jelson,jonh@microsoft.com

Abstract

Mugshot is a system that captures every event in an executing JavaScript program, allowing developers to deterministically replay past executions of web applications. Replay is useful for a variety of reasons: failure analysis using debugging tools, performance evaluation, and even usability analysis of a GUI. Because Mugshot can replay every execution step that led to a failure, it is far more useful for performing root-cause analysis than today's commonly deployed client-based error reporting systems—core dumps and stack traces can only give developers a snapshot of the system after a failure has occurred.

Many logging systems require a specially instrumented execution environment like a virtual machine or a custom program interpreter. In contrast, Mugshot's client-side component is implemented entirely in standard JavaScript, providing event capture on *unmodified* client browsers. Mugshot imposes low overhead in terms of storage (20-80KB/minute) and computation (slowdowns of about 7% for games with high event rates). This combination of features—a low-overhead library that runs in unmodified browers—makes Mugshot one of the first capture systems that is practical to deploy to every client and run in the common case. With Mugshot, developers can collect widespread traces from programs in the field, gaining a visibility into application execution that is typically only available in a controlled development environment.

1 Introduction

Despite developers' best efforts to release high quality code, deployed software inevitably contains bugs. When failures are encountered in the field, many programs record their state at the point of failure, e.g., in the form of a core dump, stack trace, or error log. That snapshot is then sent back to the developers for analysis. Perhaps the best known example is the Windows Error Reporting framework, which has collected over a billion error reports from user programs and the kernel [14].

Unfortunately, isolated snapshots only tell part of the story. The root cause of a bug is often difficult to determine based solely on the program's state after a problem was detected. Accurate diagnosis often hinges on an understanding of the events that *preceded* the failure. For this reason, systems like Flight Data Recorder [29], DejaVu [5], and liblog [13] have implemented *deterministic program replay*. These frameworks log enough information about a program's execution to replay it later under the watchful eye of a debugging tool. With a few notable exceptions, these systems require a specially instrumented execution environment like a custom kernel to capture a program's execution. This makes them unsuitable for field deployment to unmodified end-user machines. Furthermore, no existing capture and replay system is specifically targeted for the unique needs of the web environment.

Mugshot's goal is to provide low-overhead, "always-on" capture and replay for web-deployed JavaScript programs. Our key insight is that JavaScript provides sufficient reflection capabilities to log client-side nondeterminism *in standard JavaScript* running on *unmodified browsers*. As a user interacts with an application, Mugshot's JavaScript library logs explicit user activity like mouse clicks and "behind-the-scenes" activity like the firing of timer callbacks and random number generation. When the application fetches external objects like images, a server-side Mugshot proxy stores the binary data so that at replay-time, requests for the objects can access the log-time versions.

The client-side Mugshot log is sent to the developer in response to a trigger like an unexpected exception being caught. Once the developer has the log, he uses Mugshot's replay mode to recreate the original JavaScript execution on his unmodified browser. Like the logging library, the replay driver is implemented in

standard JavaScript. The driver provides a "VCR" interface which allows the execution to be run in near-real time, paused, or advanced one event at a time. The internal state of the replaying application can be inspected using unmodified JavaScript debugging tools like Firebug [17]. Developers can also analyze a script's performance or evaluate the usability of a graphical interface by examining how real end-users interacted with it.

At first glance, Mugshot's logging and replay capabilities may seem to introduce a fundamentally new threat to user privacy. However, Mugshot does not create new techniques for logging previously untrackable events—instead, it leverages the preexisting introspective capabilities found in browsers' standard JavaScript engines. Furthermore, the preexisting security policies which prevent cross-site data exchange also prevent Mugshot's event logs from leaking across domains. Thus, the Mugshot log for a particular page can only be accessed by that page's domain.

The rest of the paper is organized as follows. In Section 2, we review related work in capture and replay. We then describe the architecture of Mugshot in Section 3. Section 4 contains our evaluation, which describes microbenchmarks (§4.1) and Mugshot's performance inside several complex, real-world applications (§4.2). Our evaluation shows that Mugshot is unobtrusive at logging time and faithful at replay time, recreating several bugs that we found in our evaluation applications. We consider the privacy implications of Mugshot in Section 5 and then conclude in Section 6.

2 Related Work

Error Reporting from the Field

There are many frameworks for collecting information about crashed programs and sending the data back to the developers. Perhaps the best known is Windows Error Reporting (WER), which has been included in Windows since 1999 and has collected billions of error reports [14]. When a crash, hang, installation failure, or other error is detected, WER creates a *minidump*. Minidumps are snapshots of the system's essential state: register values, thread stacks, lists of loaded modules, a portion of the text segment surrounding the instruction pointer, and other information. With the user's permission, this data is sent back to Microsoft, where it is bucketed according to likely root-cause for later analysis. WER only records the state of the application at the moment of the crash; developers must infer the sequence of events that led to it. Other deployed systems have similar capabilities and constraints, including Firefox's Breakpad [4] and the iPhone OS [1].

Capture and Replay

As described by the survey papers [8] and [10], deterministic replay has been studied for many years in a variety of contexts. Some of the prior systems are machine-wide replay frameworks intended to debug operating systems or cache coherence in distributed shared memory systems [11, 22, 29]. They create high-fidelity execution logs, but with significant cost: the target software must run atop custom hardware, a modified kernel, or a special virtual machine. This limits opportunities for widespread deployment. Furthermore, these systems produce log data at a high rate; generally, these systems can only report the last few seconds of system state before an error.

Moving up the stack, a number of application- or language-specific tools allow deterministic capture and replay. By restricting themselves to high-level interfaces and a single-threaded execution model, they obviate the need to log data at the instruction level. This dramatically reduces the overhead of logging, both in processor time and storage requirements. The liblog system[13] is perhaps closest to our work. liblog provides a C-library interposition layer that records the input and output of all interactions between the application and the C library (and hence, the operating system) below. As with Mugshot, one of liblog's goals was to make logging sufficiently lightweight that it can be run in the common case. Other application-specific logging environments include DejaVu [5] for Java programs and Retrospect [3] for parallel programs written using the MPI interface.

Ripley [20] is a framework for preserving the computational integrity of AJAX applications. Client and server code is written in `.NET`, but Ripley automatically translates the client-side portion into JavaScript for execution in a browser. The instrumented JavaScript sends an event stream to the web server, which replays the events to a server-side replica of the client. The server only executes a client RPC if it is also generated by the replica.

Mugshot differs from Ripley in three important ways. First, Mugshot works on arbitrary JavaScript applications and does not require applications to be developed in a special environment. Second, Ripley's current implementation does not capture all sources of nondeterminism. For example, Ripley does not handle calls to `Date()`. It could treat `Date()` as an RPC and have the client synchronously fetch a value from the server (a value which would also be fed to the server-side client replica). However, this incurs a round trip for each time request, making it infeasible for applications like games that rapidly generate events. Third, for performance reasons, Ripley's server-side client replicas are not actual web browsers—they are lightweight browser emulators that track DOM state (§3.1.3) but do not perform layout

or rendering. In contrast, Mugshot replays events inside the same browser type used at logging time. This greatly increases the likelihood that a buggy execution can be recreated.

Debugging for Web Applications

There are a variety of tools for debugging web applications. For example, Fiddler [21] is a web proxy that allows the local user to inspect, modify, and replay HTTP messages. Firebug [17], an extension to Firefox, is an advanced JavaScript debugger that supports breakpoints, arbitrary expression evaluation, and performance profiling. Internet Explorer 8 has a built-in debugger with similar features. All of these tools provide rich introspection upon the local execution environment. However, none of them provide a way to capture remote bugs in the wild and explore the execution paths that led to faulty behavior.

AjaxScope [18] uses a web proxy to dynamically instrument JavaScript code before sending it to remote clients. Developers express their debugging intent through functions inserted at specific places in the code's abstract syntax tree. For example, to check for infinite loops, a programmer can attach a diagnostic function to each `for`, `while`, and `do-while` statement. Whereas AjaxScope's goal is to let developers express specific debugging policies, Mugshot focuses on recreating entire remote execution contexts.

The commercial Selenium [27] Firefox extension records user activity for later playback. Recording can only be done in Firefox, but playback is portable across browsers using synthetic JavaScript events. Because Selenium does not log the full set of nondeterministic events, it is suitable for automating tests, but it cannot reproduce many nondeterministic bugs.

The commercial products ClickTale [7] and CS SessionReplay [12] capture mouse and keyboard events in browser-based applications. However, neither of these products expose a full, browser-neutral environment for logging *all* sources of browser nondeterminism, including both client-side nondeterminism like timer interrupts and server-side nondeterminism like dynamic image generation. The services provide click analytics and a movie of client-visible interactions, but not the underlying internal state of the JavaScript heap and the browser DOM tree.

3 Design and Implementation

Mugshot's goal is to record the execution of a web application on an end user's machine, then recreate that execution on a developer's machine. To capture application activity, one could exhaustively record every intermediate state of the program. Mugshot instead takes the approach of many other systems: recording all sources of *nondeterminism*. If an application is run again and injected with the same nondeterministic events, the program will follow the same execution path that was observed at logging time.

Past systems have recorded nondeterminism at the instruction level [11] or the `libc` level [13]. However, the former may introduce prohibitive logging overheads, and both require users to modify standard browsers or operating systems. Both approaches also record nondeterminism at a granularity that is unnecessarily fine for JavaScript-driven web applications. JavaScript programs are *non-preemptively single threaded* and *event driven*. Applications register callback functions for events like key strokes or the completion of an asynchronous HTTP request. When the browser detects such an event, it invokes the appropriate application handlers. The browser will never interrupt the execution of one handler to run another. Thus, the execution path of the application is solely determined by the event interleavings encountered during a particular run. This means that logging the *content* and the *ordering* of events provides sufficient information for replay.

Logging nondeterminism at the level of JavaScript events would be easy if we could insert logging code directly into the browser. However, this solution is unappealing to developers since it requires users to download a special browser or install a logging plug-in. Many users will not opt into such a scheme, dramatically reducing the size and diversity of the developer's logging demographic.

To avoid these problems, we implemented the client portion of Mugshot entirely in JavaScript. Compared to an in-browser solution, a JavaScript implementation is more complex and more difficult to make performant. However, it has the enormous advantage of being transparent to users and hence much easier to deploy. As we will see in the sections that follow, JavaScript offers sufficient introspection and self-modification capabilities to enable insertion of shims that log most sources of nondeterminism.

In Section 3.1, we enumerate the sources of nondeterminism in web applications and describe how Mugshot captures them in Firefox and IE. Although conceptually straightforward, the logging process is complicated by various browser incompatibilities and implementation deficiencies, particularly with respect to keyboard events. In Section 3.2, we describe how Mugshot replays an execution by dispatching synthetic events from its log.

3.1 Capturing Nondeterministic Events

To add Mugshot recording to an application, the developer delivers an application through a server-side web proxy. The proxy's first job is to insert a single tag

at the beginning of the application's <head> block:

```
<script src='Mugshot.js'></script>
```

When the page loads, the Mugshot library runs before the rest of the application code has a chance to execute. Mugshot interposes on the sources of nondeterminism that we describe below and begins to write to an in-memory log. Event recording continues until the page is closed.

If the application contains multiple frames, the proxy injects the Mugshot <script> tag into each frame. Child frames report all events to the Mugshot library running in the topmost frame; this frame is responsible for collating the aggregate event log and sending it back to the developer.

The developer controls when the application uploads event logs. For example, the application may post logs at predefined intervals, or only if an unexpected exception is thrown. Alternatively, the developer may add an explicit "Send error report" button to the application which triggers a log post.

Figure 1 lists the various sources of nondeterminism in web applications. In the sections below, we discuss each of the broad categories and describe how we capture them on Firefox and IE. Our discussion proceeds in ascending order of the technical difficulty of logging each event category.

Importantly, Mugshot does *not* log events for media objects that are opaque to JavaScript code. For example, Mugshot does not record when users pause a Flash movie or click a region inside a Java applet. Since these objects do not expose a JavaScript-accessible event interface, Mugshot can make no claims about their state. The current implementation of Mugshot also does not capture nondeterministic events arriving from opaque containers like Flash's `ExternalInterface`; such events are rarely used in practice.

For each new event that it does capture, Mugshot creates a log entry containing a sequence number and the wall clock time. The entry also contains the event type and enough type-specific data to recreate the event at replay time. For example, for keyboard events, Mugshot records the GUI element that received the event, the character code for the relevant key, and whether any of the shift, alt, control, or meta keys were simultaneously pressed.

3.1.1 Nondeterministic Function Calls

Applications call `new Date()` to get the current time and `Math.random()` to get a random number. To log time queries, Mugshot wraps the original constructor for the `Date` object with one that logs the returned time. To log random number generation, Mugshot replaces the built-in `Math.random()` with a simple linear congruential generator [23]. Mugshot uses the application's load date to seed the generator, and it writes this seed to the log. Given this seed, subsequent calls to the random number generator are deterministic and do not require subsequent log entries.

Our initial implementation of Mugshot did not define a custom random number generator—instead, it simply wrapped `Math.random()` in the same way that it wrapped `Date()`. However, we found that games often made frequent requests for random numbers, e.g., to determine whether a space invader should move up or down. The resulting logs were filled with random numbers and did not compress well (which was important, since uncompressed logs can be large (§ 4.2.1)). Thus, we decided to use the logging scheme described above.

3.1.2 Interrupts

JavaScript interrupts allow applications to schedule callbacks for later invocation. Callbacks can be scheduled for one-time execution using `setTimeout(callback, waitTime)`. A callback can be scheduled for periodic execution using `setInterval(callback, period)`. JavaScript is cooperatively single threaded, so interrupt callbacks (and event handlers in general) execute atomically and do not preempt each other.

Mugshot logs interrupts by wrapping the standard versions of `setTimeout()` and `setInterval()`. The wrapped registration functions take an application-provided callback, wrap it with logging code, and register the wrapped callback with the native interrupt scheduler. Mugshot also assigns the callback a unique id; since JavaScript functions are first class objects, Mugshot stores this id as a property of the callback object. Later, when the browser invokes the callback, the wrapper code logs the fact that a callback with that id executed at the current wall clock time.

Although simple in concept, IE does not support this straightforward interposition on `setTimeout()` and `setInterval()`. Mugshot's modified `setTimeout()` must hold a reference to the browser's original `setTimeout()` function; however, IE sometimes garbage collects this reference, leading to a "function not defined" error from the Mugshot wrapper. To mitigate this problem, Mugshot creates an invisible <iframe> tag, which comes with its own namespace and hence its own references to `setTimeout()` and `setInterval()`. The Mugshot wrapper invokes copies of these references when it needs to schedule a wrapped application callback.

Although this trick gives Mugshot references to the native scheduling functions, it prevents Mugshot from actually scheduling callbacks until the hidden frame

	Event type	Examples	Captured by Mugshot
DOM Events §3.1.3	Mouse	click, mouseover	Yes
	Key	keyup, keydown	Yes
	Loads	load	Yes
	Form	focus, blur, select, change	Yes
	Body	scroll, resize	Yes
Interrupts §3.1.2	Timers	setTimeout(f, 50)	Yes
	AJAX	req.onreadystatechange = f	Yes
Nondeterministic functions §3.1.1	Time query	(new Date()).getTime()	Yes
	Random number query	Math.random()	Yes
Text selection §3.1.8	Firefox: window.getSelection()	Highlighting text w/mouse	Yes
	IE: document.selection	Highlighting text w/mouse	Partially
Opaque browser objects §3.1	Flash movie	User pauses movie	No
	Java applet	Applet updates the screen	No

Figure 1: Sources of nondeterminism in browsers.

is loaded. This problem has three cascading consequences. First, since JavaScript is single-threaded, Mugshot cannot block until the hidden frame is loaded without hanging the application. Instead, it must queue application timer requests and install them once the hidden frame loads. Second, `setTimeout()` and `setInterval()` return opaque scheduling identifiers that the application can use to cancel the callback via `clearTimeout()` and `clearInterval()`. For interrupt registrations issued before the hidden frame loads, Mugshot cannot call the real registration functions to get cancellation ids. So, Mugshot generates synthetic identifiers and maintains a map to the real identifiers it acquires later. Third, an application may cancel an interrupt before the hidden frame loads; Mugshot responds by simply removing the callback from its queue of requests.

AJAX requests allow JavaScript applications to issue asynchronous web fetches. The browser represents each request as an *XMLHttpRequest* object. To receive notifications about the status of the request, applications assign a callback function to the object's `onreadystatechange` property. The browser invokes this function whenever new data arrives or the entire transmission is complete. Upon success or failure, the various properties of the object contains the status code for the transfer (e.g., `200 OK`) and the fetched data.

Mugshot must employ different techniques to wrap AJAX callbacks on different browsers. On Firefox, Mugshot's wrapped `XMLHttpRequest` constructor registers a DOM Level 2 handler (§3.1.3) for the object's `onreadystatechange` event. IE does not support DOM Level 2 handlers on AJAX objects, so Mugshot interposes on the object's `send` method to wrap the application handler in logging code before the browser issues the request. We describe DOM Level 2 handlers in more detail in the next section.

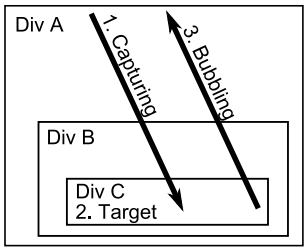
For each AJAX event, Mugshot logs the current state of the request (e.g., "waiting for data"), the HTTP headers, and any incremental data that has already returned. Once the request has completed, Mugshot logs the HTTP status codes. We also log the raw request data on the server-side replay proxy (§3.2.1). For the purposes of replay, this data only needs to be logged on one side. Thus, our current implementation consumes more space than strictly necessary. However, it makes the client-side and proxy-side logs more understandable to human debuggers, since AJAX activity in one log does not have to be collated with data from the other log.

3.1.3 DOM Events

The Document Object Model (or DOM) is the interface between JavaScript applications and the browser's user interface [28]. Using DOM calls, JavaScript applications register handlers for user events like mouse clicks. DOM methods also allow the application to dynamically modify page content and layout.

The browser binds every element in a page's HTML to an application-accessible JavaScript object. Applications attach event handlers to these DOM objects, informing the browser of the application code that should run when a DOM node generates a particular event. In the simplest handler registration scheme, applications simply assign functions to specially-named DOM node properties. For example, to execute code whenever the user clicks on a `<div>` element, the application assigns a function to the `onclick` property of the corresponding JavaScript DOM node.

This simple model, called DOM Level 0 registration, only allows a single handler to be assigned to each DOM node/event pair. Modern browsers also implement the DOM 2 model, which allows an application to register multiple handlers for a particular DOM node/event pair.




```
<div onclick='handlerA()'>
   <div onclick='handlerB()'>
      <div onclick='handlerC()'>
      </div>
   </div>
</div>
```

Figure 2: Event handling after a user clicks within Div C. In the W3C model's capturing phase, handlerA() is called if it is a capturing handler, followed by handlerB() if it is a capturing handler. After the target's handlerC() is called, W3C mandates a bubbling phase in which handlers marked as bubbling are called from the inside out.

An application calls the node's `attachEvent()` (IE) or `addEventListener()` (Firefox) method, passing an event name like "click", a callback, and in Firefox, a `useCapture` flag to be discussed shortly.

The World Wide Web Consortium's DOM Level 2 specification [28] defines a three-phase dispatch process for each event (Figure 2). In the *capturing* phase, the browser hands the event to the special `window` and `document` JavaScript objects. The event then traces a path down the DOM tree, starting at the top-level `<html>` DOM node and eventually reaching the DOM node that actually generated the event. The boolean parameter in `addEventListener()` allows a handler to be specified as capturing. Capturing handlers are only executed in the capturing phase; they allow a DOM node to execute code when a child element has generated an event. Importantly, the ancestor's handler will be called *before* any handlers on the child run.

In the *target* phase, the event is handed to the DOM node that generated it. The browser executes the appropriate handlers at the target, and then sends the event along the reverse capturing path. In this final *bubbling* phase, ancestors of the target can run event handlers marked as bubbling, allowing them to process the event after it has been handled by descendant nodes.

In the DOM 2 model, some event types are cancelable, i.e., an event handler can prevent the event from continuing through the three phase process. Also, although all events capture, some do not bubble. For example, `load` events, which are triggered when an image has completely downloaded, do not bubble. Form events also do not bubble. Examples of form events include `focus` and `blur`, which are triggered when a GUI element like a text box gains or loses input focus.

3.1.4 DOM Events and Firefox

Firefox supports the W3C model for DOM events. Thus, Mugshot can record these events in a straightforward way—it simply attaches capturing logging handlers to the `window` object. Since the `window` object is the highest ancestor in the DOM event hierarchy, Mugshot's logging code is guaranteed to catch every event before it has an opportunity to be canceled by other nodes in the capture, target, or bubble phases. Note that canceled events still need to be logged, since they caused at least one application handler to run!

Mugshot must ensure that the application does not accidentally delete or overwrite Mugshot's logging handlers. To accomplish this, Mugshot registers the logging handlers as DOM 2 callbacks, exploiting the fact that applications cannot iterate over the DOM 2 handlers for a node, and they cannot deregister a DOM 2 handler via `domNode.detachEvent(eventName, callback)` without knowing the callback's function pointer.

Mugshot must also ensure that its DOM 2 `window` handlers run before any application-installed `window` handlers execute and potentially cancel an event. Firefox invokes a node's DOM 2 callbacks in the order that they were registered; since the Mugshot library registers its handlers before any application code has run, its logging callbacks are guaranteed to run before any application-provided DOM 2 `window` handlers.

Unfortunately, Firefox invokes any DOM 0 handler on the node before invoking the DOM 2 handlers. To ensure that Mugshot's DOM 2 handler runs before any application-provided DOM 0 callback, Mugshot uses JavaScript setters and getters to interpose on assignments to DOM 0 event properties. Setters and getters define code that is bound to a particular property on a JavaScript object. The setter is invoked on read accesses to the property, and the getter is invoked on writes.

Ideally, Mugshot would define a DOM 2 logging handler for each event type `e`, and create setter code for the `window.e` property which wrapped the the user-specified handler with a Mugshot-provided logging function. If the application provided no DOM 0 handler, Mugshot's DOM 2 callback would log the event; otherwise, the wrapped DOM 0 handler would log the event and set a special flag on the event object indicating that Mugshot's DOM 2 handler should not duplicate the log entry. Unfortunately, this scheme will not work because Firefox's getter/setter implementation is buggy. Mugshot can create a getter/setter pair for a DOM node event prop-

erty, and application writes to the property will properly invoke the setter. However, when an actual event of type `e` is generated, the browser will not invoke the associated function. In other words, the setter code, which works perfectly at the application level, hides the event handler from the internal browser code.

Luckily, the setter code does not prevent the browser from invoking DOM 2 handlers. Thus, Mugshot's setter also registers the application-provided handler as a DOM 2 callback. The setter code ensures that when the application overwrites the DOM 0 property name, Mugshot deregisters the shadow DOM 2 version of the old DOM 0 handler.

When Mugshot logs a DOM event, it records an identifier for the DOM node target. If the target has an HTML id, e.g., `<div id='foo'>`, Mugshot tags the log entry with that id. Otherwise, it identifies the target by specifying the capturing path from the root `<html>` tag. For example, the id $(1, 5)$ specifies that the target can be reached by following the first child of the `<html>` tag and then the fifth child of that node. Since many JavaScript applications use dynamic HTML, the path for a particular node may change throughout a program's execution. Thus, the determination of a target's path id must be done at the time the event is seen—it cannot be deferred until (say) the time that the log is posted to the developer.

3.1.5 DOM Events and IE

IE's event model is only partially compatible with the W3C one. The most important difference is that IE does not support the capturing phase of event propagation. This introduces two complications. First, an event may never have an opportunity to bubble up to a `window`-level logging handler—the event might be canceled by a lower-level handler, or it may be a non-bubbling event like a load. Second, even if an event bubbles up to a `window`-level logger, the event may have triggered lower-level event handlers and generated loggable nondeterministic events. For example, a mouse click may trigger a target-level callback that invokes `new Date()`. The mouse click is temporally and causally antecedent to the time query. However, the mouse click would be logged *after* the time query, since the time query is logged at its actual generation time, whereas the mouse click is logged after it has bubbled up to the `window`-level handler.

Mugshot addresses these problems using several techniques. To log non-bubbling events, Mugshot exploits IE's facility for extending the object prototypes for DOM nodes. For DOM types like `Images` and `Inputs` which support non-bubbling events, Mugshot modifies their class definitions to define custom setters for DOM 0 event properties. Mugshot also redefines `attachEvent()` and `detachEvent()`, the mechanisms by which applications register DOM 2 handlers for these nodes. The DOM 0 setters and the wrapped DOM 2 registration methods collaborate to ensure that if an application defines at least one handler for a DOM node/event pair, Mugshot will log relevant events precisely once, and before any application-specified handler can cancel the event.

Ideally, Mugshot could use the same techniques to capture bubbling events at the target phase. Unfortunately, IE's DOM extension facility is fragile: redefining certain combinations of DOM 0 properties can cause unpredictable behavior. Therefore, Mugshot uses `window`-level handlers to log bubbling events; this is the problematic technique described above that may lead to temporal violations in the log. Fortunately, Mugshot can mitigate this problem in IE, because IE stores the current DOM event in a global variable `window.event`. Whenever Mugshot needs to log a source of nondeterminism, it first checks whether `window.event` is defined and refers to a not-yet-logged event. If so, Mugshot logs the event before examining the causally dependent event.

An application may cancel a bubbling event before it reaches Mugshot's `window`-level handler by setting its `Event.cancelBubble` property to `true`. The event still must be logged, so Mugshot extends the class prototype for the `Event` object, overriding its `cancelBubble` setter to log the event before its cancellation.

In summary, Mugshot on IE logs all bubbling DOM events, but only the non-bubbling events for which the application has installed handlers. This differs from Mugshot's behavior on Firefox, where it logs all DOM events regardless of whether the application cares about them. Recording these "spurious" events does not affect correctness at replay time, but it does increase log size. Fortunately, as we show in Section 4.2.1, Mugshot's compressed logs are small enough that the storage penalty for spurious events is small. Thus, we were not motivated to implement an IE-style logging solution for Firefox—it was technically feasible, but comparatively much more difficult to implement correctly than our capturing-handler solution.

3.1.6 Handling Load Events on IE

In IE, `load` events do not capture or bubble. Using the techniques described in the previous section, Mugshot can capture these events for elements with application-installed `load` handlers. However, Mugshot actually needs to capture *all* load events so that at replay time, it can render images in the proper order and ensure that

the page layout unfolds in the same fashion observed at logging time. Otherwise, an application that introspects the document layout may see different intermediate results at replay time.

Ideally, Mugshot could modify the prototype for `Image` objects such that whenever the browser created an `Image` node, the node would automatically install a logging handler for `load` events. Unfortunately, prototype extension only works for properties and methods accessed by application-level code—the browser's native code creation of the DOM node cannot be modified by extending the JavaScript-level prototype. So, Mugshot uses a hack: whenever it logs an event, it schedules a timeout to check whether that event has created new `Image` nodes; if so, Mugshot explicitly adds a DOM 2 logging handler which records load events for the image. Mugshot specifies this callback by invoking a non-logged `setTimeout(imageCheck, 0)`. The 0 value for the timeout period makes the browser invoke the `imageCheck` callback "as soon as possible." Since the timeout is set from the context of an event dispatch, the browser will invoke the callback immediately after the dispatch has finished, but before the dispatch of other queued events (such as the `load` of an image that we want to log). Mugshot also performs this image check at the end of the initial page parse to catch the loading of the page's initial set of images.

3.1.7 Synthetic Events

Applications call `DOMnode.fireEvent()` on IE and `DOMnode.dispatchEvent()` on Firefox to generate synthetic events. Mugshot uses these functions at replay time to simulate DOM activity from the log. However, the application being logged can also call these functions. These synthetic events are handled synchronously by the browser; thus, from Mugshot's perspective, they are deterministic program outputs which do not need to be logged. However, in terms of the event dispatching path, the browser treats the fake events just like real ones, so they will be delivered to Mugshot's logging handlers.

To prevent these events from getting logged on Firefox, Mugshot interposes on `document.createEvent()`, which applications must call to create the fake event that will be passed to `dispatchEvent()`. The interposed `document.createEvent()` assigns a special `doNotLog` property to the event before returning it to the application. Mugshot's logging code will ignore events that define this property.

This technique does not work on IE, which prohibits the addition of new properties to the Event object. Thus, Mugshot uses prototype extension to interpose on `fireEvent()`. Inside the interposed version, Mugshot pushes an item onto a stack before calling the native `fireEvent()`. After the call returns, Mugshot pops an item from the stack. In this fashion, if Mugshot's logging code for DOM events notices a non-empty stack, it knows that the current DOM event is a synthetic one and should not be logged.

3.1.8 Annotation Events

At replay time, Mugshot dispatches synthetic DOM events to simulate user GUI activity. These events are indistinguishable from the real ones with respect to the dispatch cycle—given a particular application state, a synthetic event will cause the exact same handlers to execute in exactly the same order as a semantically equivalent real event. However, we noticed that synthetic events did not always update the visible browser state in the expected way. In particular, we found the following problems on both Firefox and IE:

- According to the DOM specification, when a `keypress` event has finished the dispatch cycle, the target text input or content-editable DOM node should be updated with the appropriate key stroke. Our replay experiments showed that this did not happen reliably. For example, synthetic key events could be dispatched to a text entry box, but the value of the box would not change, despite the fact that the browser invoked all of the appropriate event handlers.
- `<select>` tags implement drop-down selection lists. Each selectable item is represented by an `<option>` tag. Dispatching synthetic mouse clicks to `<option>` nodes should cause changes in the selected item property of the parent `<select>` tag. Neither Firefox nor IE provided this behavior.
- Users can select text or images on a web page by dragging the mouse cursor or holding down the shift key while tapping a directional key. The browser visibly represents the selection by highlighting the appropriate text and/or images. The browser internally represents the selected items as a range of underlying DOM nodes. Applications access this range by calling `window.getSelection()` on Firefox and inspecting the `document.selection` object on IE. We found that dispatching synthetic key and mouse events did not reliably update the browser's internal selection range, and it did not reliably recreate the appropriate visual highlighting.

To properly replay these DOM events, Mugshot defines special *annotation events*. Annotation events are "helpers" for events which, if replayed by themselves, would not produce a faithful recreation of the logging-

time application state. Mugshot inserts an annotation event into the log immediately after a DOM event which has low fidelity replay. At replay time, Mugshot dispatches the low fidelity synthetic event, causing the appropriate event handlers to run. Mugshot then executes the associated annotation event, finishing the activity induced by the prior DOM event. Annotation events are not real events, so they do not trigger application-defined event handlers. They merely describe work that Mugshot must perform at replay time to provide faithful emulation of logging-time behavior.

To fix low-fidelity `keypress` events on text inputs, Mugshot's `keypress` logger schedules a timeout interrupt with an expiration time of 0. The browser executes the callback immediately after the end of the dispatch cycle for the `keypress`, allowing Mugshot to log the value of the text input. At replay time, after dispatching the synthetic `keypress`, Mugshot reads the value annotation from the log and programmatically assigns the value to the target DOM node's `value` property.

To ensure that clicks on `<option>` elements actually update the chosen item for the parent `<select>` tag, Mugshot's `mouseup` logger checks whether the event target is an `<option>` tag. If so, this indicates that the user has selected a new choice. Mugshot generates an annotation indicating which of the `<select>` tag's children was clicked upon. At replay time, Mugshot uses the annotation to directly set the `selectedIndex` property of the `<select>` tag.

Mugshot generates annotation events for selection ranges after logging `keyup` and `mouseup` events. On Firefox, the selection object conveniently defines a starting DOM node, a starting position within that node, an ending DOM node, and an ending position within that node. Mugshot simply adds the relevant DOM node identifiers and integer offsets to the annotation record. Abstractly speaking, Mugshot includes the same information for an annotation record on IE. However, IE does not provide a straightforward way to determine the exact extent of a selection range. So, Mugshot must cobble together several IE range primitives to deduce the current range. Mugshot first determines the highest enclosing parent tag for the currently selected range. Then, Mugshot creates a range which covers all of the parent tag's children, and progressively shrinks the number of HTML characters it contains, using IE's `Range.inRange()` to determine whether the actual selection region resides within the shrinking range. At some point, Mugshot will determine the exact amounts by which it must shrink the left and right margins of the parent range to precisely cover the actual selected region. Mugshot logs the DOM identifier for the parent node and the left and right pinch margins.

IE's selection semantics are extremely complex, and we have not produced a complete formal specification for them. Since Mugshot cannot currently reproduce these semantics in all applications, Figure 1 lists Mugshot's support for IE selection events as partial.

3.1.9 Performance Optimizations

Both Firefox and IE support the W3C `mousemove` event, which is fired whenever the user moves the mouse. Mugshot can log this event like any other mouse action, but this can lead to unnecessary log growth in Firefox if the application does not care about this event (remember that Mugshot on Firefox logs all DOM events, regardless of application interest in them). Mugshot logs `mousemove` by default, but since few applications use `mousemove` handlers, the developer can disable its logging to reduce log size. In Section 4.2, we evaluate Mugshot's log size for a drawing application that *does* use `mousemove` events.

Games which have high rates of keyboard or mouse activity may generate many content selection annotation events. Generating these annotations is expensive on IE since Mugshot has to experimentally determine the selection range (see Section 3.1.8). Furthermore, games do not typically care about the selection zones that their GUI inputs may or may not have created. Thus, for games with high event rates like Spacius [16], we disable annotations for content selection.

3.2 Replay

Compared to the logging process, replay is straightforward. The most complexity arises from replaying load events, since JavaScript code cannot modify the network stack and stall data transmissions to recreate logging-time load orderings. Thus, Mugshot coordinates load events with a transparent caching proxy that the developer inserts between his web server and the outside world.

In addition, Mugshot must also shield the replaying execution from *new* events that arise on the developer machine, e.g., because the developer accidentally clicks on a GUI element in the replaying application. Without a barrier for such new events, the replaying program may diverge from the execution path seen at logging time.

3.2.1 Caching Web Content at Logging Time

When a user fetches a page logged by Mugshot, the fetch is mediated by a transparent Mugshot proxy. The proxy assigns a session ID to each page fetch; this ID is stored in a cookie and later written to the Mugshot log. As the proxy returns content to the user, it updates a

per-session cache which maps content URLs to the data that was served for those URLs during that particular session. Optionally, the proxy can rewrite static `<html>` and `<frame>` declarations to include the Mugshot library's `<script>` tag.

3.2.2 Replaying Load Events

At replay time, the developer switches the proxy into replay mode, sets the session ID in his local Mugshot cookie to the appropriate value, and directs his web browser to the URL of the page to replay. The proxy extracts the session ID from the cookie, determining the cache it should use to serve data. The proxy then begins to serve the application page, replacing any static `<script>` references to the logging Mugshot library to references to the Mugshot replay library.

During the HTML parsing process, browsers load and execute `<script>` tags synchronously. The Mugshot replay library is the first JavaScript code that the browser runs, so Mugshot can coordinate load interleavings with the proxy before any load requests have actually been generated. During its initialization sequence, Mugshot fetches the replay log from the developer's log server and then sends an AJAX request to the proxy indicating that the proxy should only complete the loads for subsequent non-`<script>` objects in response to explicit "release load" messages from Mugshot.

The rest of the page loads, with any `<scripts>` loading synchronously. The browser may also launch asynchronous requests for images, frame source, etc. These asynchronous requests queue at the proxy. Later, as the developer rolls forward the replay, Mugshot encounters load events for which the corresponding browser requests are queued at the server. Before signaling the proxy to transmit the relevant bytes, Mugshot installs a custom DOM 2 `load` handler for the target DOM node so that it can determine when the load has finished (and thus when it is safe to replay the next event).

3.2.3 The Replay Interface

At replay initialization time, Mugshot places a semi-transparent `<iframe>` overlaying the application page. This frame acts as a barrier for keyboard and mouse events, preventing the developer from issuing events to the replaying application that did not emerge from the log. We embed a VCR-like control interface in the barrier frame which allows the developer to start or stop replay (see Figure 3). The developer can single-step through events or have Mugshot dispatch them at fixed intervals. Mugshot can also try to dispatch the events in real time, although "real-time" playback of applications with high event rates may have a slowdown factor of 2 to 4 times (see Section 4.2.2).

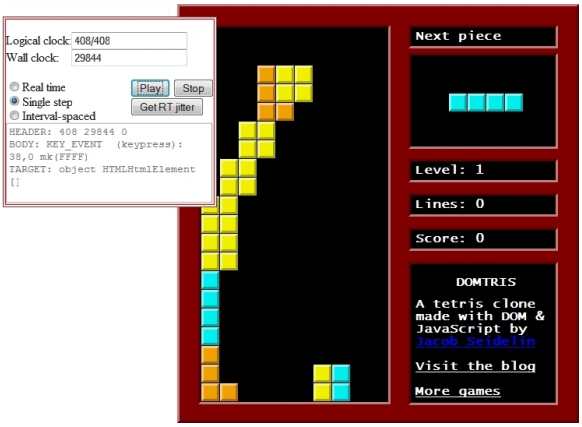


Figure 3: Replaying Tetris (VCR control on left)

Whenever Mugshot replays an event, it can optionally place a small, semi-translucent square above the target DOM node. These squares are color-coded by event type and fade over time. They allow the developer to visually track the event dispatch process, and are particularly useful for understanding mouse movements.

3.2.4 Replaying Non-`load` Events

Replaying events is much simpler than logging them. To replay a non-`load` DOM event, Mugshot locates the target DOM node and dispatches the appropriate synthetic event. For low-fidelity events (§3.1.8), Mugshot also performs the appropriate fix-ups using annotation records. To replay text selection events, Mugshot recreates the appropriate range object and then uses a browser-specific call to activate the selection.

To replay timeout and interval callbacks, Mugshot's initialization code interposes on `setTimeout()` and `setInterval()`. The interposed versions tag each application-provided callback with an interrupt ID and add the callback to a function cache. This cache is built in the same order that IDs were assigned at logging time, so replay-time interrupt IDs are guaranteed to be faithful. Mugshot does not register the application-provided callback with the native interrupt scheduler. Instead, when the log indicates that an interrupt should fire, Mugshot simply retrieves the appropriate function from its cache and executes it. Mugshot interposes on the cancellation functions `clearTimeout()` and `clearInterval()`, but the interposed version are no-ops—once the application cancels an interrupt at logging time, Mugshot will never encounter it again in the log.

Mugshot also interposes on the `XMLHttpRequest` constructor. Much like interrupt replay, Mugshot stores AJAX callbacks in a function cache and executes them at the appropriate time. Mugshot updates each synthetic

AJAX object with the appropriate log data before invoking the application AJAX handler.

By interposing on the `Date()` constructor, Mugshot forces time queries to read values from the log. The log also contains the initialization seed used by the random number generator at capture time. Mugshot uses this value to seed the replay-time generator. This is sufficient to ensure that subsequent calls to `Math.random()` replay faithfully.

3.3 Limitations

Mugshot uses a caching proxy to reproduce the load events in the log. If an application fetches external content that does not pass through the proxy, Mugshot cannot guarantee faithful replay of its data or its load time. Thus, these ill-defined loads can ruin the fidelity of the entire replay.

As described in Section 3.1.8, Mugshot must use annotation records to properly replay GUI events involving drop-down boxes. Although the replay is correct from the perspective of a `<select>` tag's internal JavaScript state, both IE and Firefox refuse to visually drop-down a drop-down list in response to synthetic events. However, after Mugshot applies the annotation event, the visual display of the tag adjusts to indicate the appropriate selection.

Web applications typically fail because of unexpected interactions between HTML, CSS, and event-driven JavaScript code. Mugshot logs all of these application inputs and the associated event streams. In many cases, this is sufficient to recreate a bug; the log-time browser need not be the exact same type and version as the replay-time browser. However, some bugs arise from an interaction between the application and a *specific* browser type and version. In these cases, it is crucial for the log-time and replay-time browsers to be the same. Mugshot's client-side component records the identity of its log-time browser (e.g., Firefox 3.5) so that the developer can run the same browser at debug time. However, even this may be insufficient to recreate some bugs—users can install browser plug-ins or change local configuration state, and the existence of that particular local state may be the root cause of a bug. Since this type of client state cannot be introspected by JavaScript code, Mugshot cannot reliably reproduce these kinds of bugs.

If a web page contains multiple frames, proper logging requires each frame to contain the Mugshot logging `<script>` tag. Similarly, at replay time, each frame must contain Mugshot's replay script. The replay proxy can automatically instrument statically declared frames. However, if a page dynamically creates frames, e.g., using JavaScript, the developer is responsible for inserting the appropriate Mugshot tags.

4 Evaluation

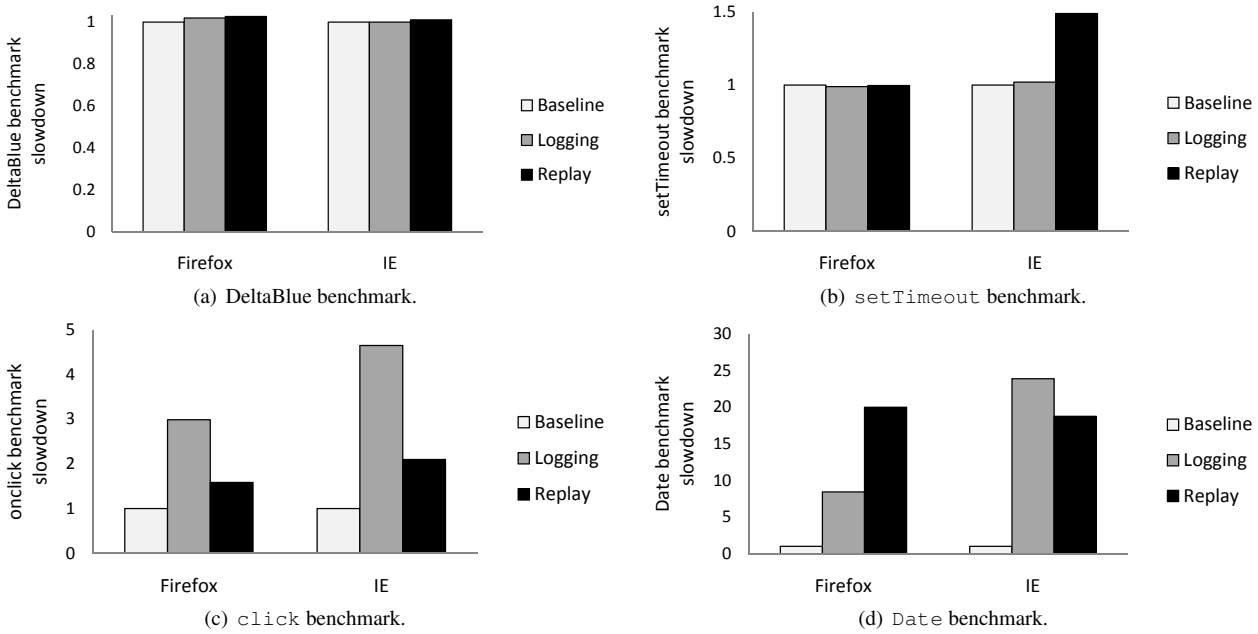
For Mugshot to be useful, it must be unobtrusive at logging time and faithful to the original program execution at replay time. If event logging makes programs sluggish, users will reject Mugshot-enabled applications; if Mugshot cannot reproduce real bugs, it provides no utility to application developers. In this section, we run Mugshot on a variety of microbenchmarks and real JavaScript programs, demonstrating that user-perceived logging overhead is no worse than 6.8% for applications with high event rates. We provide two examples of bugs that Mugshot can log and then reproduce at replay time. We also demonstrate that replay speed is not unacceptably slow, and that events logs grow no faster than 100 KB per minute in applications with high event rates.

All experiments ran on an HP xw46000 workstation with a dual-core 3GHz CPU and 4 GB of RAM. We tested Mugshot performance inside two browsers, Firefox v3.5.3 and IE8 v8.0.6001. When stripped of extraneous white space and comments, Mugshot's logging code was 46 KB and its replay code was 35 KB. Note that only the logging code must be shipped to end users, and only the replay code must be shipped to debugger machines.

4.1 Microbenchmarks

To explore the basic computational overheads of logging and replay, we inserted Mugshot into several microbenchmark applications. For each microbenchmark, we compared its run time without Mugshot support to its run time during logging and replay. We used the following test suite:

- DeltaBlue is a constraint solver from Google's V8 JavaScript benchmark suite [15]. The benchmark is computationally intensive, but it has no user interface, and it does not internally generate loggable events. Thus, DeltaBlue's Mugshot-enabled running time reflects any penalty that Mugshot imposes on straightline computational workloads.
- The `Date` benchmark simply called `new Date()` 5000 times.
- In baseline and logging mode, the `click` benchmark dispatched 5000 synthetic mouse events as quickly as possible. As explained in Section 3.1.7, Mugshot normally does not log synthetic GUI events since they are deterministic. However, for the logging part of the benchmark, we forced Mugshot to log the synthetic mouse clicks. At replay time, Mugshot simply tried to dispatch these logged events as quickly as possible.
- The `setTimeout` benchmark issued 25 calls to `setTimeout(function(){}, 0)`. If the computational overheads of logging and replaying a null function are high, fewer interrupts can be issued per unit time.



(a) DeltaBlue benchmark.

(b) `setTimeout` benchmark.

(c) `click` benchmark.

(d) `Date` benchmark.

Figure 4: Microbenchmark slowdown due to logging and replay.

Figure 4 shows the Mugshot-induced slowdowns for the test suite. Each graph depicts a benchmark's execution time in baseline, logging, and replay scenarios; performance is normalized with respect to baseline performance. Each result represents the average of 10 trials, and in all cases, standard deviations were less than 5%.

As expected, Figure 4(a) shows that Mugshot introduced no overhead for a purely computational workload. Figure 4(b) demonstrates that logging activity did not delay interrupt scheduling in Firefox and IE. However, replay did introduce a 50% slowdown on IE; we are still investigating the reasons for this behavior.

As shown in Figure 4(c), logging penalties slowed the `click` benchmark by a factor of 3 on Firefox and 4.6 on IE. The overhead primarily arose from the complex logic needed to properly log mouse events on `<select>` elements (Section 3.1.8). During replay, the `click` benchmark slowed by a factor of 1.6 on Firefox and 2.1 on IE. Most of the slowdown was caused by regular expression computations during the parsing of the each `click` log entry. An optimized version of Mugshot would parse the entire log at replay initialization time. However, as we show in Section 4.2, our unoptimized Mugshot can already replay real applications at a tolerable rate.

Figure 4(d) shows the Mugshot penalties for the `Date` microbenchmark. The slowdown factors are large, ranging from 8.4 to 23.9. The reason is that fetching the current date in the baseline case is extremely fast— it merely requires a read of a native browser variable. User-level JavaScript code is much slower than native code, so Mugshot's `Date()` logging introduces high relative overheads. Fortunately, Section 4.2 shows that real applications do not issue time queries at a high enough rate to expose Mugshot's logging overhead.

4.2 Application Examples

To evaluate Mugshot's performance in more realistic conditions, we examined its logging and replay overheads for seven applications. Three of the applications were games with varying rates of event generation.

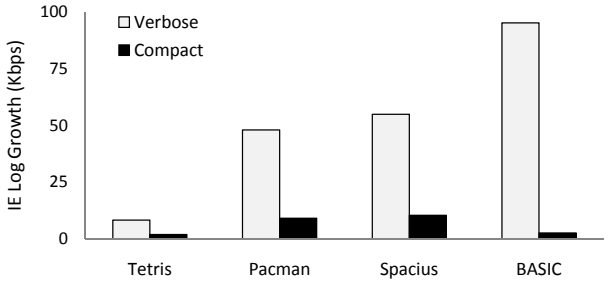
- DOMTRIS [26] is a JavaScript implementation of the classic Tetris game.
- Pacman [6] is unsurprisingly a Pacman clone.
- Spacius [16] is a 2D side-scrolling space shooter.

These games implement many of their animations using interrupt callbacks, so frame rates (and the user experience) will suffer if Mugshot introduces too much latency to the critical path of interrupt dispatch.

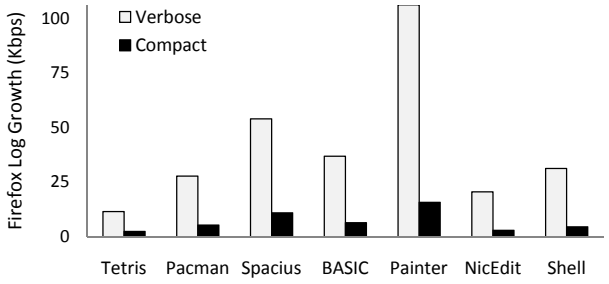
We also evaluated Mugshot's performance on four non-games:

- The Applesoft BASIC interpreter [2] parses and runs BASIC programs, providing an emulated joystick and graphical display.
- NicEdit [19] is a WYSIWYG text and HTML editor.
- Painter [24] is a simple drawing program.
- The JavaScript shell [25] provides a command-line interface for manipulating the DOM and application-defined JavaScript state.

These programs stress Mugshot's handling of form, key, and selection events. Painter also makes use of the

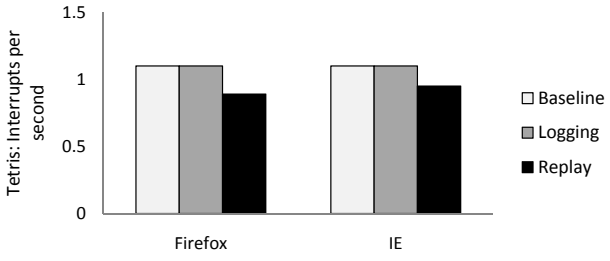


(a) IE applications.

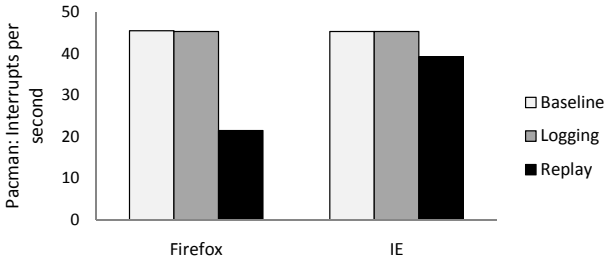


(b) Firefox applications.

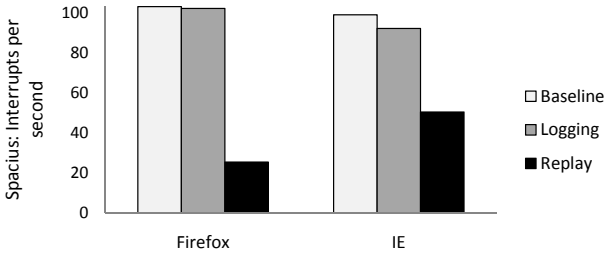
Figure 5: Growth rate of logs (kilobits per second).



(a) Tetris.



(b) Pac-man.



(c) Spacius.

Figure 6: Interrupt rates.

`mousemove` event, so we configured Mugshot to log those events for this application.

We evaluated Mugshot's performance in the Firefox browser for each of the seven applications. However, we only evaluated Mugshot's IE performance for the first four applications. The latter three applications do not replay correctly in IE; they trigger quirks in IE's form/selection event model that Mugshot does not currently handle.

4.2.1 Log Sizes

Figure 5 depicts the growth rate of Mugshot's log for each application, showing the size of the verbose log and the compact log. The verbose log has a human-friendly format; among other things, it contains a dump of the page's HTML at load time, and it explicitly tags each event with an easy-to-understand string representing the event type and its parameters. In our experiences, just reading the verbose log can provide a human debugger with invaluable insights about program operation. The compact log discards the beautifications of the verbose log and represents events and their parameters using short status codes. The compact log is also compressed using the LZW algorithm with a window size of 200.

Figure 5 shows that the rates of uncompressed log growth varied widely, from roughly 10 Kbps (Tetris) to 106 Kbps for Painter on Firefox and 95 Kbps for the BASIC interpreter on IE. Figure 5 also shows that Mugshot's log compression is effective, with a worst case compressed growth rate of 15.7 Kbps for the Painter application. Painter's logs were comparatively large because Mugshot had to log frequent `mousemove` events. Spacius generated the second highest growth rates (10.9 Kbps KB on Firefox and 10.4 Kbps on IE). As we discuss in the next section, this was due to Spacius' high rate of interrupt events.

4.2.2 Logging and Replay Overheads

For Mugshot to be practical, its logging overhead must have a minimal impact on the user experience. It is less important for Mugshot to be able to replay application traces in real time. However, replaying should not be so slow that debugging is painful for a developer.

Games update the screen in response to GUI events like mouse clicks. However, for graphically intense games, most of the screen updates are driven by timer interrupt callbacks. The dispatch rate of these callbacks provides a natural metric for Mugshot-induced slowdowns—the more overhead that Mugshot creates, the slower these callbacks execute, and the more sluggish the application appears.

Figure 6 shows the interrupt dispatch rate for the three games on IE and Firefox. As with the microbenchmark results, we show dispatch rates for baseline, logging, and replay scenarios. To measure these rates, we manually identified all of the interrupt handlers in each game, adding a single line of code to each handler which incremented a global counter. At the end of 30 seconds, we divided this counter by the elapsed wall time to get the number of interrupts dispatched per second.

Figures 6(a) and 6(b) show that for applications with low to moderate interrupt rates, the interrupt dispatch rate was unchanged, i.e., Mugshot logging introduced no overhead. Compared to Tetris and Pacman, Spacius had a very high interrupt rate, executing about 100 callbacks per second. Figure 6(c) shows that for this application, Mugshot's logging overhead reduced dispatch rates by 0.8% on Firefox and 6.8% on IE. However, Spacius gameplay did not seem qualitatively degraded during logging on either browser.

Figure 6 shows that dispatch rates at replay time can decrease by as much as 75% in the case of a Spacius replay on Firefox. This time dilation is certainly tolerable, but as mentioned in Section 4.1, we could improve the replay rate by optimizing our log parsing.

4.2.3 Capturing Real Bugs

Mugshot's goal is to capture application runs and replay them on developer machines. An important application of replay mode is the recreation of buggy application states. Armed with the event interleavings that generate a program fault, a developer can use powerful localhost debuggers to step through the log and inspect the application's state after each event.

Since we lacked detailed changelogs for the seven applications described in Section 4.2, we could not intentionally undo a bug fix and then see whether Mugshot could successfully log and replay a problematic event sequence. However, while performing the experiments in Section 4.2, we did encounter bugs in two of the applications, both of which Mugshot could log and replay.

The first bug involved a display glitch in the Tetris program which we were able to characterize in detail using Mugshot. A Tetris game terminates if a falling block nestles amongst the static blocks in a way that causes the overall block structure to exceed a maximum allowable height. When this happens, the game should render the bottom part of the most recent piece but leave the top part clipped, since this part extends above the playable area. However, depending on the shape of the most recent piece and the preexisting block structure, the Tetris implementation we tested would incorrectly render the final block structure, scattering the constituent blocks of the final piece in arbitrary positions, sometimes overwriting preexisting blocks. Figure 3 shows an example of this bug. The final piece is 1 block by 4 blocks, but when its stacking causes the game to end, two of its blocks mysteriously materialize in the square formation at the bottom of the screen.

The second bug involved the Painter application. To draw a rectangle in this program, the user clicks on the "Rectangle Tool", then drags the pointer across the canvas with the mouse button down. If the user selects the "Rectangle Tool" and just single-clicks on the drawing area, no rectangle should be drawn. However, after single clicking, an expanding rectangle will appear as the user moves the mouse. This makes the user think that he is, in fact, drawing a rectangle. However, when the mouse is clicked again, the rectangle suddenly disappears.

We captured, replayed, and diagnosed both bugs using Mugshot. For both applications, the compressed log which captured the bug was under 11 KB in size. Transmitting such an error report to developers would be extremely fast, even on a slow connection.

5 Privacy

Mugshot provides developers with an extremely detailed log of user behavior. Some might worry that this leads to an unacceptable violation of user privacy. Such privacy concerns are valid. However, web sites can (and should) provide an "opt-in" policy for Mugshot logging, similar to how Windows users must willingly decide to send performance data to Microsoft [14].

We also emphasize that Mugshot is not a fundamentally new threat to online privacy. Web developers already have the ability to snoop on users to the extent allowed by JavaScript, and to send the resulting data back to their own web servers. Indeed, many web sites already perform a crude version of event logging using web analysis services like CrazyEgg [9] that build heat maps of click activity on a particular page. In all cases, the scope of JavaScript-based snooping is limited by the browser's cross-site scripting policies. From the browser's perspective, Mugshot is not an exception: it is subject to exactly the same restrictions designed to thwart malware. These restrictions prevent all programs—including Mugshot—from snooping on a frame owned by one domain and sending that data to a different domain.

6 Conclusions

As web applications have grown in popularity, browsers have shipped with increasingly powerful JavaScript debuggers. These tools are extremely useful for introspecting applications that are running on a local

development machine. However, they cannot be used to examine program contexts which reside on remote machines. When regular end users encounter application bugs, they will not inspect the application using their browser's advanced debugger. At best, they will send a bug report which describes their problem using natural language. At worst, they will do nothing and simply be frustrated. Ideally, users would have a convenient way to give developers the precise event sequence that led to a buggy application state. The developer could then recreate the execution run and use his knowledge of the code to diagnose the problem.

To address these issues, we created Mugshot, a lightweight framework for capturing JavaScript application runs and replaying them on different machines. Experiments show that Mugshot introduces little overhead at logging time. For applications like games which generate many events, Mugshot slows execution speeds by 6.8% in the worst case. Mugshot event logs grow at a reasonable rate, requiring 20–80 KB per minute of application activity. Using Mugshot's replay mode, we have successfully recreated bugs in two real applications. Mugshot's logs also support usability investigations and traditional click analytics.

References

[1] APPLE COMPUTER. Crash Reporting for iPhone OS Applications, 2009. Technical Note TN2151.

[2] BELL, J. Applesoft BASIC Interpreter in Javascript. http://www.calormen.com/applesoft/.

[3] BOUTEILLER, A., BOSILCA, G., AND DONGARRA, J. Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2007), vol. 4757 of *Lecture Notes in Computer Science*, Springer, pp. 297–306.

[4] BREAKPAD. http://kb.mozillazine.org/Breakpad. MozillaZine Knowledge Base.

[5] CHOI, J.-D., AND SRINIVASAN, H. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools* (1998), pp. 48–59.

[6] CIESLAK, K. PacMan! http://www.digitalinsane.com/api/yahoo/pacman/.

[7] CLICKTALE LTD. ClickTale: Record, Watch, Understand. http://www.clicktale.com, 2009.

[8] CORNELIS, F., GEORGES, A., CHRISTIAENS, M., RONSSE, M., GHESQUIERE, T., AND BOSSCHERE, K. D. A taxonomy of execution replay systems. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet* (2003).

[9] CRAZY EGG, INC. CrazyEgg: Visualize Your Visitors. http://crazyegg.com/, 2010.

[10] DIONNE, C., FEELEY, M., AND DESBIENS, J. A Taxonomy of Distributed Debuggers Based on Execution Replay. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (1996), pp. 203–214.

[11] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of OSDI* (2002), pp. 211–224.

[12] FORESEE RESULTS, INC. CS SessionReplay. http://www.4cresults.com/CSSessionReplay.html, 2009.

[13] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *Proceedings of USENIX Technical* (2006), pp. 289–300.

[14] GLERUM, K., KINSHUMANN, K., GREENBERG, S., AUL, G., ORGOVAN, V., NICHOLS, G., GRANT, D., LOIHLE, G., AND HUNT, G. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of SOSP* (2009), pp. 103–116.

[15] GOOGLE. V8 JavaScript benchmarks. http://code.google.com/apis/v8/benchmarks.html.

[16] HACKETT, M., AND MORSE, J. Spacius. http://scriptnode.com/lab/spacius/.

[17] JON HEWITT. Ajax Debugging with Firebug. http://http://www.ddj.com/architect/196802787, January 10, 2007.

[18] KICIMAN, E., AND LIVSHITS, B. AjaxScope: A Platform for Remotely Monitoring the Client-side Behavior of Web 2.0 Applications. In *Proceedings of SOSP* (2007), ACM, pp. 17–30.

[19] KIRCHOFF, B. NicEdit. http://nicedit.com/.

[20] K.VIKRAM, PRATEEK, A., AND LIVSHITS, B. RIPLEY: Automatically Securing Web 2.0 Applications Through Replicated Execution. In *Proceedings of CCS* (2009), pp. 173–186.

[21] LAWRENCE, E. Fiddler: Web Debugging Proxy. http://www.fiddler2.com/fiddler2, 2009. Microsoft Corporation.

[22] NARAYANASAMY, S., POKAM, G., AND CALDER, B. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro 26*, 1 (2006), 100–109.

[23] PARK, S., AND MILLER, K. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM 31* (1988), 1192–1201.

[24] ROBAYNA, R. Canvas Painter. http://caimansys.com/painter/.

[25] RUDERMAN, J., MIELCZAREK, T., LEE, E., AND RONN-JENSEN, J. JavaScript Shell. http://www.squarefree.com/shell/.

[26] SEIDELIN, J. DOMTRIS. http://www.nihilogic.dk/labs/tetris/.

[27] SELENIUM WEB APPLICATION TESTING SYSTEM. http://seleniumhq.org/, 2009.

[28] WOOD, L., NICOL, G., BYRNE, S., CHAMPION, M., HORS, A. L., HÉGARET, P. L., AND ROBIE, J. Document object model (DOM) level 2 core specification, Nov. 2000. http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.

[29] XU, M., BODÍK, R., AND HILL, M. D. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *Proceedings of ISCA* (2003), pp. 122–133.