Fine-grained access-control for the Puppet configuration language

Bart Vanbrabant, Joris Peeraer, Wouter Joosen
bart.vanbrabant@cs.kuleuven.be, jorispeeraer@gmail.com, wouter.joosen@cs.kuleuven.be
DistriNet, Dept. of Computer Science,
K.U.Leuven, Belgium

Abstract

System configuration tools automate the configuration and management of IT infrastructures. However these tools fail to provide decent authorisation on configuration input. In this paper we apply fine-grained authorisation of individual changes on a complex input language of an existing tool. We developed a prototype that extracts meaningful changes from the language used in the Puppet tool. These changes are authorised using XACML. We applied this approach successfully on realistic access control scenarios and provide design patterns for developing XACML policies.

1 Introduction

The management of large IT infrastructures needs to be automated to keep it manageable and reduce the amount of human errors [3, 11]. A system configuration tool is software that enables a system administrator to automate the configuration and management of large IT infrastructures. These tools address scalability, heterogeneity, and the consistency of relations between machines [2]. All system configuration tools have a similar reference architecture: each managed device runs an agent that manages the configuration of that device. The agent compares the current state of the device with the state described in a policy that is stored in a database or repository on a central server. This policy determines the configuration and the state of the entire IT infrastructure. Therefore if someone adds unauthorised changes to the central policy this person can control the entire IT infrastructure. Thus access control to this central policy is required.

System configuration tools can be divided into two categories based on how the input policy is organised: database or textual [5] based. The database based tools often use a graphical interface or command interface to manipulate their policies. Access control in these tools is enforced on records in that database. The other input

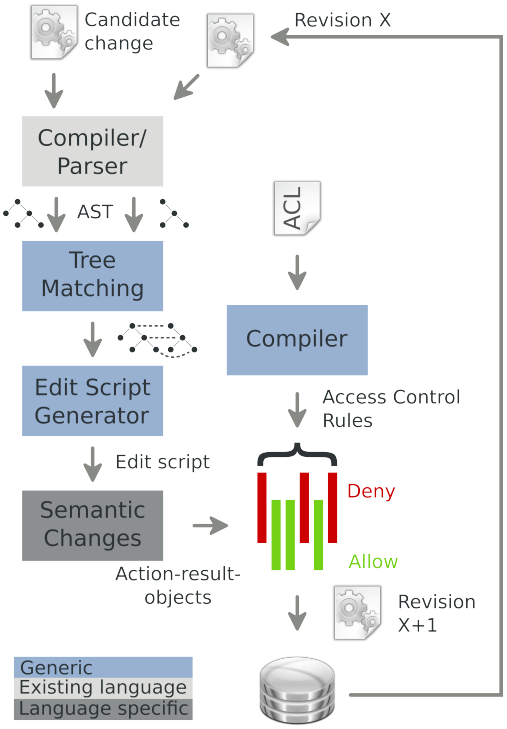

Figure 1: Overview of the solution presented in this paper.

type uses textual configuration files. The current state-of-practice of these text based system configuration tools uses path based access control to prevent unauthorised access to the textual configuration files but the name and the path of the file often do not have a relation with the contents of the file. To be able to use conventional path based access control, current tools rely on conventions to determine in which file what configuration statement may be included. For example network related configuration can only be defined in the `network.cf` configuration file. System management tools and path based

access control however cannot prevent a malicious user from adding network configuration statements to the file `motd.cf`.

In "Federated Access Control and Workflow Enforcement in Systems Configuration" [10] we proposed a method called ACHEL to enforce fine-grained access control based on the semantics of a change. In other words ACHEL calculates the operations the user wants to authorise using the changes in a textual file. We applied this approach to a minimal configuration language to prove its viability in a prototype. This prototype used a custom access control language based on regular expressions. Our method is language agnostic, except for the part to give meaning to each change. Figure 1 show the steps in the ACHEL method.

In this paper we apply the ACHEL method on a configuration language of an existing system configuration tool. The two research objectives of this paper are:

1. Can we extract the AST from the compiler of a system configuration tool and can we reuse this internal AST or do we need to transform it in order to obtain differences that are semantically meaningful?

2. How do we authorise changes? Once changes are known they have to be authorised. We propose an access control language and design patterns in this paper that provide the flexibility to express the different rules in a manageable and understandable fashion.

In this paper we add fine-grained access control based on meaningful changes to Puppet [1] and used XACML [8] to authorise changes. We implemented a prototype and integrated it with a version control system. Afterwards we evaluated this prototype by comparing it with traditional access control in two change scenarios. An important subset of the Puppet language is supported and we present design patterns to use the full expressiveness of the Puppet language including the unsupported language constructs with our access control mechanism.

In the remainder of this paper we first give some background on the ACHEL authorisation mechanism in Section 2. Then we look at related work in Section 3. Afterwards we discuss the methods used in the differencing process in Section 4. In this section we also discuss the problems we encountered by using the AST Puppet generates. The next step is the actual authorisation. Section 5 introduces the XACML framework and proposes a design pattern for writing policies. Finally in Section 6 we compare our method with other authorisation methods.

2 The ACHEL authorisation mechanism

The configuration model used by a system management tool is compiled from an input in the form of textual source code. This source input is stored on a filesystem or in a repository that uses version tracking. Access control and authorisation in the state of the art is based on operations performed on files and directories. In state of the art system management tools there is often no link between the file path and the parts of the configuration model represented in the file. Version control systems use diff-like algorithms [9] that operate on flat files to generate changes between two versions of a file. Diff algorithms detect changed lines and produce a list of insert and remove line operations. Applying access control on these operations does not make much sense. The operations are highly syntax dependant and there is only a weak link between the insert and remove operations and the configuration model.

In large infrastructures updates are never applied directly to the production infrastructure. Depending on the contents of the update or the person that produced the update, different authorisations can be required. For example:

1. all changes from junior administrators need to be reviewed and approved by a senior administrator

2. the scenario in Figure 2 where a change needs to be approved by a manager

3. all changes to the production infrastructure outside maintenance windows require approval by two managers

4. in a federated infrastructure changes to the backbone network need to be approved by the management of each of the administrative domains

Existing system management tools and access control solutions provide no support for these complex workflows.

Our method [10] transforms the updates on the configuration model by comparing the current and the new version of the input source. It compiles the two versions to an abstract syntax tree [7]. From the two versions an *edit script* is generated that transforms the old AST to the AST of the new version [4]. This process is represented in Figure 1. Because we are working on the AST, we know the semantics of changes made to the nodes in the abstract syntax tree. Therefore the edit script can be transformed to operations on entities that exist in the configuration model. Using our method, access control rules can be expressed in terms of operations on the entities in the configuration model. For example, instantiating a new resource, instead of adding a line to the input file that has a given syntax. These operations are the actions

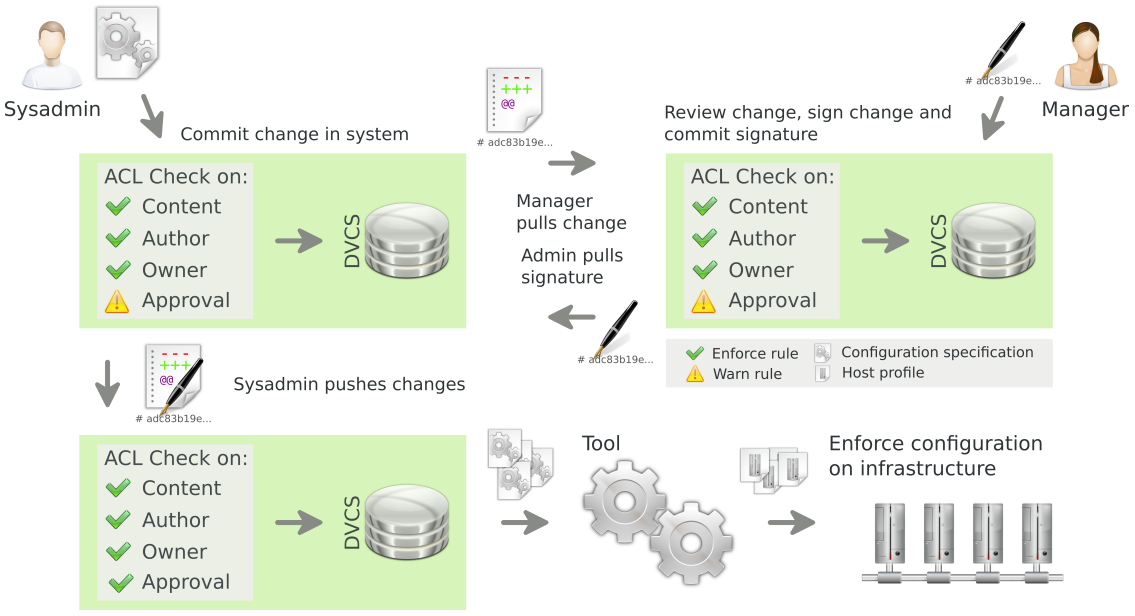

Figure 2: Updating the configuration model using access control.

that needs to be authorised. Additionally this method, opposed to other access control methods, derives the operations to authorise automatically and requests permission to apply them.

For audit purposes the configuration model is often stored in version controlled repositories. These repositories record each change to a configuration file and metadata such as the user that made the change and an optional log message. In ACHEL changes to this repository are digitally signed with the private key of the administrator. During generation of the edit script and the transformation of the edit script, the owner of each entity and the author of each change is tracked. The owner of an entity is the user that added or modified the entity. This ownership and author information is also exposed to the access control engine.

We enforce update workflows by using distributed version control repositories. Each system administrator that makes changes to the configuration model has their own repository. Distributed version control repositories assign a unique identifier to each change based on the contents of the change. To enforce update workflows, a change is authorised by the owner of a key by signing this unique identifier and including it as an update in the repository. Access control rules can require the authorisation of a third party before an update is allowed. Because each distributed version control repository can have its own set of access control rules, very flexible update workflows can be enforced.

Figure 2 represents a possible scenario supported by ACHEL [10]. A system administrator makes a change that is allowed in his repository but it requires approval by a manager to push the change into the repository for the production infrastructure. The sysadmin requests the manager to review his change. The manager reviews the change and approves it by signing the identifier of the change. The sysadmin can now push his change to the production repository together with the signature of the manager.

3 Related work

In "A survey of system configuration tools" [5] we evaluated several system configuration tools, including their support for access control and authorisation of changes. We identified two types of authorisation: either path based access control or access control based on "resources" in the configuration model. The tools that support external version repositories can reuse the path based access control of that repository or the access control models that the filesystem provides. Other tools allow fine grained access control on "resources" in a database using a hierarchy of resources. The system configuration tools that enforce authorisation on "resources" do this on resources in the configuration model that is used to generate and deploy configuration files and manage each system. The main disadvantage of this method is that authorisation cannot be performed on language constructions that are determined at runtime. For example the usage of a *Collect* instruction in Puppet.

"Authorisation and Delegation in the Machination Configuration System" [6] proposes a method of organ-

ising and delegating access to configuration information. The author integrated this method in the configuration management tool Machination. One of the key requirements of his method is the ability to authorise access to configuration aspects individually. He accomplishes this requirement by authorising the primitive operations which manipulate the configuration. The configuration representation used in Machination, is a form of XML with additional restrictions. These restrictions assure every configuration element is addressable by an XPath query. Upon this representation, a set of primitive operations is defined. These operations edit the configuration by adding, removing, changing and ordering the individual elements in the configuration input. Authorisation is then performed upon the individual operations needed to transform the configuration. By grouping multiple elements together such that they can be referred to by an XPath query, multiple configuration aspects can be authorised.

Both tools use the principle of authorisation on the individual elements. Where Machination starts from one version and uses the operation to obtain the new version, ACHEL derives the operations that need to be authorised from the two versions. This paper describes a method in which the differencing of ACHEL is used to find the changes made to a file.

4 Extracting changes

The ACHEL authorisation method starts from the configuration file that has been changed. The method consists of two phases. The first phase retrieves the AST of each file from the Puppet compiler. This AST should not contain any grammatical constructs anymore for the differencing algorithm to work. This is not the case for the AST Puppet produces, it still contains some syntactical leftovers. Therefore a transformation step is also included in this phase. The second phase compares the two trees and calculates an edit script that describes the operation to transform the first AST into the second AST. This edit script is transformed into meaningful changes expressed in terms of the language constructs in the Puppet language.

The Puppet language is an expressive language that also contains control flow and runtime evaluated expression such as the case statement or virtual and exported resources. Applying access control to changes that include these language constructs is very hard with our method because the effects of such statements are only known when the "configuration policy" of each managed device is calculated. In this prototype of the ACHEL authorisation method for Puppet we support a limited set of language constructs in the Puppet language on which we can apply authorisation. This set includes creating definitions, classes, creating resources including using arrays as identifier and relations. In Section 5.2 we will argue why this limited set can already be used to create powerful access control policies.

4.1 Generating the abstract syntax tree

The ACHEL authorisation mechanism requires access to the AST of each version of the Puppet manifests. In this section we describe how we extract the AST from Puppet. The AST from Puppet is not directly usable for our mechanism because it contains syntactical constructions. Therefore we need to normalise this AST.

We use the Puppet parser to create the AST of a Puppet input file. This provides us the AST that Puppet reasons upon. However this AST is not suited for generating an edit script. Although it is an *abstract* syntax tree, the tree still contain syntactical language constructs from the Puppet language. This is a problem because they do not have any meaning. This can even result in two different AST's that have the same semantics. Consider Figure 3a and the corresponding AST in Figure 3b. Line 1 in Figure 3a describes the declaration of two users. The AST of the code fragment still contains the array which is nothing more than syntactic sugar to easily create two resources with the same attributes. For the differencing algorithm the only difference is the addition of one string to an array, instead of adding an entire resource. This change is not meaningful and cannot be described correctly in a policy.

The solution is to transform and normalise this tree to remove all syntactical structures from the abstract syntax tree. In the example from the previous paragraph we can remove the array as identifier for the users, and replicate the whole definition of the user for each element of this array. This transformation ensures that when a user gets added or removed, the differencing will detect a user being added or removed. The transformed AST is depicted in Figure 3c.

The solution in this example is very specific for the given problem and there is no generic solution to remove the syntax leftovers in the AST or even to detect them. Moreover, we do not have a list of problematic structures that are present in the Puppet language. The only solution to fully support a language is to test every language concept and check the resulting AST structure. In this implementation of our authorisation mechanism we explicitly chose to transform the AST instead of running a preprocessor over the input source. With this approach we reuse the existing lexer and parser of the Puppet tool. This makes the transformation step less syntax dependant. If a concept results in an ambiguous AST or one that contains syntactical constructs, the AST needs to be transformed or the compiler needs to be adapted.

```
user {["kwik","kwak"]:
  gid => 123
}
```

(a) The Puppet manifest

```
class: ASTClass
- member: Resource
  + type: Name => user
  + title: ASTArray
  | + child: String => kwik
  | - child: String => kwak
  - parameter: ResourceParam
    + param: Name => gid
    - value: String => 123
```

(b) The AST created by Puppet

```
class: ASTClass
+ member: Resource
| + type: String => user
| + title: String => kwik
| - parameter: ResourceParam
|   + param: Name => gid
|   - value: String => 123
- member: Resource
  + type: String => user
  + title: String => kwak
  - parameter: ResourceParam
    + param: Name => gid
    - value: Name => 123
```

(c) The normalised AST

Figure 3: Puppet configuration that defines multiple users using one resource definition.

The differencing stage compares the two normalised AST's to generate an edit script. In this prototype we use the same algorithm as our previous work [10]. This algorithm works as follows:

1. Match the leaves of the two trees using a similarity function.

2. Match the internal nodes using the information of already matched leaves: nodes with a lot of leaves in their subtrees in common are likely to match as well.

3. Correct wrongly coupled leaves using information of the matched internal nodes: parents of matching leaves should match as well.

4. Generate an edit script with the basic changes: add, modify and delete.

5. Correct changes: e.g. remove changes that cancel each other.

4.2 Generating meaningful changes

Authorisation is enforced based on operations derived from the meaning of a change and not on the operations the operations in an edit script, therefore the edit script is transformed in meaningful changes. For instance, the mode parameters of the of the `/etc/motd` changed from 0600 to 0644 instead of the 0600 node in the AST was removed and replaced by the 0644 node. These meaningful changes express changes as operations on the concepts that exist in the Puppet configuration language, instead of operations on a tree. This step is language dependant. The edit script expresses operations on the nodes in the AST. These nodes in the AST are linked to specific concepts in the Puppet language. In this step a transformation between the operations and the AST nodes and possible operations on language constructs is required. In our method this is a manually coded step.

5 Authorising changes

The second component in our solution is the authorisation of individual configuration changes. For this authorisation two elements are needed: a set of policies describing which changes are allowed or denied and a framework that executes the actual authorisation. XACML provides both features and is widely used authorisation standard in industry. Therefore we used it for implementing the authorisation step. In this section we will discuss the use of XACML to describe the access control policies.

5.1 The XACML standard

XACML is a international standard for access control and authorisation. The standard defines a language for policies and a language for authorisation requests. Both are XML based. The standard also describes the components and the architecture of an authorisation engine and allows an XACML authorisation engine to be extended.

XACML defines the components and the dataflow between them in the authorisation engine. The following components are required to handle an authorisation request:

- **Policy Enforcement Point (PEP)** This component receives the authorisation requests and creates a XACML request from it that is sent to the PDP.
- **Policy Decision Point (PDP)** The PDP loads all required policies and validates the request from the PEP against these policies. The results of these checks are combined and sent back to the PEP.
- **Policy Access Point (PAP)** The PAP makes the policies available to the PDP.

- **Policy Information Point (PIP)** The PIP provides the PDP with attributes related to subject, resource or environment. These attributes can be retrieved from several sources such as files or databases.

XACML is a generic solution for domain specific authorisation. The domain specific entities involved in the authorisation process can be mapped to subject, resource and action from the XACML standard. The subject submits a request to perform an action on a resource. Each of these entities can have multiple attributes. A policy decision is based on these attributes and additional attributes provided by the PIP.

The policy contains the rules that define what is allowed. Policies can be grouped in policy sets and each policy set can consist of policies and other policy sets. A policy is built from targets, rules, a rule combination algorithm and a number of obligations.

- The target of a policy defines when a policy needs to be used. This is expressed using a matching expression over the attributes of subject, action and resource.

- Rules have a target that defines when a rule is applicable, a condition and an effect that defines *Permit* or *Deny* based on the condition.

- The combining algorithm determines what the final result of a policy is if multiple rules returned an effect.

- The obligation is an action that needs to be executed when a policy is applicable. The PEP is responsible for executing these obligations.

The authorisation process works by exchanging request and response messages between the PDP and the outside world. The request message contains the subject, resource, action and environment and the associated attributes. If the content of the resource is XML it can be embedded in the request message. When the PDP has calculated the result of the authorisation a response message is sent back. This response message contains a result code and an optional message or information.

5.2 XACML policies for Puppet

Our authorisation method provides the configuration changes to the XACML engine that enforces authorisation. This section explains how a policy can reference Puppet language constructions in a configuration change. This section also provides a design pattern to encapsulate unsupported language constructions to enforce authorisation on them. The XACML standard describes an XML-based policy language. This language provides methods to access and compare attributes of the resource, subject and action involved in the authorisation-request. Complex functions can be used to process these attributes and to calculate the outcome of the policies. An example policy is shown in Figure 4.

XACML policies need a method for referring to the operations an update consists of and to the Puppet language constructs the operations act upon. Because XACML is based on XML and the configuration input is already available in the form of a abstract syntax tree. Additionally a resource can be embedded in a XACML authorisation request if the resource is represented in XML. Therefore we developed an XML serialisation of the Puppet manifests. We based this serialisation on the approach of Machination [6] to refer to constructs in the input using XPath. The AST is transformed into an XML tree that can be referenced uniquely by means of XPath expressions. Figures 5a, 5b and 5c show a Puppet statement and the two representations of the AST.

XPath queries can refer to the individual elements in the XML serialisation of the AST. When the node-id of a node is known, this attribute can be used to refer directly to this node using an XPath query like `//*[@id ="3"]`. When referring to the node in function of its attributes and location the following XPath query can be used: `//class[@name="apache"]/*[@type="package"]`

Puppet classes and definitions can be used to create abstractions on which access control can be enforced. These abstractions can encapsulate the language concepts our prototype currently does not support or that are very hard or impossible to support because of their dynamic nature. We used this design pattern in our evaluation the create access control policies. Superusers are allowed to make all changes, including the statements that are not supported. These superusers encapsulate these statements in definitions and classes that can be used by other users. This design pattern matches closely to the configuration module approach used by Puppet. These modules encapsulate the domain expert knowledge in easy to use interfaces and classes.

5.3 Using external information sources

XACML can use external sources for information through a PIP. In our prototype we extended the XACML engine to retrieve external information from directory services such as LDAP or active directory. These directories contained the roles of each user that can make changes. Storing this information in an external source and making it available in the XACML engine, makes it possible for the XACML policies to be more generic. Puppet also supports external sources for retrieving the classes that it should assign to hosts. One of such exter-

```
<Policy PolicyId="nodes:apache">
  <Target><Resources><Resource>
    <ResourceMatch MatchId="xacml:function:xpath-node-match">
      <AttributeValue DataType="xacml2:data-type:xpath-expression">
        //class[@name="apache"]
      </AttributeValue>
      <ResourceAttributeDesignator AttributeId="xacml:resource:resource-id"
        DataType="xacml2:data-type:xpath-expression" />
    </ResourceMatch>
  </Resource></Resources></Target>
  <Rule Effect="Permit" RuleId="nodes:apache:webadmin">
    <Target />
    <Condition>
      <Apply FunctionId="xacml:function:string-greater-than">
        <AttributeValue DataType="xs:string">xyz</AttributeValue>
        <Apply FunctionId="xacml:function:string-one-and-only">
          <SubjectAttributeDesignator DataType="xs:string"
            AttributeId="xacml:subject:subject-id"/>
        </Apply>
      </Apply>
    </Condition>
  </Rule>
</Policy>
```

Figure 4: A sample XACML policy file for configuration changes.

```
# Apache-class
class apache inherits webserver {
  package {"apache": ensure => installed }
}
```

(a) Sample Puppet configuration

```
Root
+ hostclasses: ResourceType ()
| - class: ASTClass (name:apache)
|   + parent: Name () => webserver
|   - member: Resource (title:apache,type:package)
|     + parameter: ResourceParam (param:ensure)
|     - value: Name () => installed
- nodes: ResourceType ()
```

(b) The abstract syntax tree

```
<Root id='1' nodetype='ASTRoot' xmlns='pupa'>
  <hostclasses id='2' nodetype='ResourceType' >
    <class id='3' nodetype='ASTClass' name='apache'>
      <parent id='4' nodetype='Name' >webserver</parent>
      <member id='5' nodetype='Resource' title='apache' type='package'>
        <parameter id='7' nodetype='ResourceParam' param='ensure'>
          <value id='8' nodetype='Name' >installed</value>
        </parameter>
      </member>
    </class>
  </hostclasses>
  <nodes id='9' nodetype='ResourceType' >
  </nodes>
</Root>
```

(c) The XML representation of the AST

Figure 5: Puppet configuration file and the resulting AST and its XML representation

nal sources is an LDAP directory. This information can also be exposed in the XACML engine through a PIP.

6 Evaluation

We evaluated our prototype based on two access control scenarios. These scenarios each describe a policy that has to be enforced. In the evaluation we construct a policy-file that tries to accomplish this task and explain the reasons behind its structure. We compare the results of our policy with a policy based on path based access control available in version control systems. The goal of these evaluations is to show the possibilities and limitations of our tool.

For this evaluation we integrated our prototype into a version control system (VCS). The VCS is used as storage for the configuration files and also acts as the authorising agent. Additionally a Policy Information Point (PIP) provides additional attributes to the XACML engine. The PIP manages information and is contacted during the authorisation process when specific attributes are needed. The PIP in our implementation connects to an LDAP server and provides attributes belonging to the administrator issuing the change.

In the evaluation, two scenarios are investigated:

1. Only system administrators that are members of the webadmin group can configure a machine as an apache webserver.

2. Only system administrators that are members of the webadmin group can create a virtual host on an apache webserver. Moreover, the documentroot of the virtual host can only exist inside the homedirectory of the user issuing the change.

6.1 Scenario 1: configure a machine as webserver

In the first scenario we have a simple Puppet configuration that configures nodes as a webserver. In the Puppet module path an apache module is added with a class named `apache`. The site.pp file contains a list of nodes that each have a list of include statements that add functionality to that server. Figure 6 shows the initial site.pp file for this scenario. The change in this scenario configures the spare server *san-jose* as webserver by including the `apache` class from the apache module. The security policy says that all administrators can add *roles* to servers by including classes, but only users from the group *webadmin* may configure a server as webserver.

A SVN repository can be used to limit access to the file in the repository. Figure 7 shows a file with access control rules for this repository. The puppet user has read-only access to the site.pp file in the site directory and has access to the files in the apache module. Only webadmin users can edit the files in the apache module, but this does not prevent them from including for example a statement that installs a dns server in that module. The site.pp file can be edited by all administrators. The access control mechanism from SVN cannot prevent non-webadmin users to create new webservers either.

Figure 8 shows the XACML policy for the ACHEL authorisation mechanism. This policy allows users from the webadmin group to include apache classes in the configuration of a node. This XACML policy provides a very fine-grained access control to the statements in the Puppet manifests.

```
1  node baltimore {
2    # include the apache module
3    include apache
4  }
5
6  node san-jose {
7    # spare machine to configure as
         webserver
8  }
```

Figure 6: The site.pp file for Puppet for the first scenario. One server is already configured as webserver. The other server is kept as spare server.

```
1  [/modules/apache]
2  puppet = r
3  @webadmin = rw
4
5  [/site]
6  puppet = r
7  @admins = rw
```

Figure 7: An authorisation file for SVN to restrict access to the Puppet manifests in the repository.

6.2 Scenario 2: add virtual hosts

The second scenario uses the same apache webserver setup and allows users from the webuser group to add virtual hosts to the apache configuration. Webusers can only add virtual hosts to the configuration from which the document root is located in their own home directory. The document root parameter controls the directory that contains the files that a webserver should serve to visitors of a particular domain. The home directory path of users is always built as follows: `/home/` and their username concatenated to that. The Puppet manifest in

```
<Policy PolicyId="nodes:apache">
  <Target><Resources><Resource>
    <ResourceMatch MatchId="xacml:function:xpath-node-match">
      <AttributeValue DataType="xacml2:data-type:xpath-expression">
        //p:node/p:include[@class="apache"]
      </AttributeValue>
      <ResourceAttributeDesignator
        AttributeId="xacml:resource:resource-id"
        DataType="xacml2:data-type:xpath-expression" />
      </ResourceMatch>
  </Resource></Resources></Target>
  <Rule Effect="Permit" RuleId="nodes:apache:webadmin">
    <Target><Subjects><Subject>
      <SubjectMatch MatchId="xacml:function:anyURI-equal">
        <AttributeValue DataType="xs:anyURI">webadmin</AttributeValue>
        <SubjectAttributeDesignator AttributeId="xacml2:subject:role" DataType="xs:anyURI
            " />
      </SubjectMatch>
    </Subject></Subjects></Target>
  </Rule>
</Policy>
```

Figure 8: The XACML policy to allow users from the webadmin group to add the apache class to a node.

Figure 9 shows a manifest from the apache module that configures the virtual hosts in the system.

The access control configuration for a SVN repository for this scenario is similar to the previous scenario. Figure 11 shows an updated authorisation file for the SVN repository that contains the Puppet manifests for this scenario. This file limits access to the vhosts.pp file to users from the webuser group. It cannot prevent users from adding virtual hosts that have a document root in the home directory of another user. It also does not prevent users from adding virtual host resources to other manifest files.

The XACML policy in Figure 10 enforces access control based on the contents of the change and not based on the file location. It enforces access control on all occurrences of virtual host resources in any Puppet manifest file that is included in the repository. The policy builds the home directory of the user by concatenating */home/* with the username. The value of the documentroot parameters of the virtual host resource should always start with the home directory string the policy created.

This policy can be circumvented by using a path that starts with the users home directory but then uses `..` to traverse back to the `/home` directory. For example, `/home/lisa/../foo/www`. This is not a flaw of the approach but a limitation of the expressiveness of the XACML functions. To close this policy a function that normalises the path first before it does the compare is required. In the conclusion we will discuss how to counter this attack.

```
class apache {
  apache::vhost {"www.example.com":
    docroot => "/home/lisa/www"
  }

  apache::vhost {"photo.example.com":
    docroot => "/home/lisa/photo"
  }
}
```

Figure 9: The vhosts.pp file for Puppet for the second scenario that is located in the /vhosts directory.

```
[/modules/apache]
puppet = r
@webadmin = rw

[/vhosts]
puppet = r
@webuser = rw

[/site]
puppet = r
@admins = rw
```

Figure 11: An update authorisation file for SVN for the second scenario.

6.3 Conclusion

The conclusions from this evaluation are twofold. First, there is a strong need for content-aware authorisation.

```
<Policy PolicyId="apache:webuser">
  <Target><Subjects><Subject>
    <SubjectMatch MatchId="xacml:function:anyURI-equal">
      <AttributeValue DataType="xs:anyURI">webuser</AttributeValue>
      <SubjectAttributeDesignator
        AttributeId="xacml2:subject:role" DataType="xs:anyURI" />
    </SubjectMatch>
  </Subject></Subjects></Target>
  <Rule Effect="Permit" RuleId="apache:webuser:vhost">
    <Description>Add or remove a vhost</Description>
    <Target><Resources><Resource>
      <ResourceMatch MatchId="xacml:function:xpath-node-equal">
        <AttributeValue DataType="xacml2:data-type:xpath-expression">
          //pup:*[@type="apache::vhost"]
        </AttributeValue>
        <ResourceAttributeDesignator AttributeId="xacml:resource:resource-id"
          DataType="xacml2:data-type:xpath-expression" />
      </ResourceMatch>
    </Resource></Resources></Target>
  </Rule>
  <Rule Effect="Permit" RuleId="apache:webuser:vhost-docroot">
    <Target><Resources><Resource>
      <ResourceMatch MatchId="xacml:function:xpath-node-match">
        <AttributeValue DataType="xacml2:data-type:xpath-expression">
          //p:*[@type="apache::vhost"]/p:parameter[@param="docroot"]
        </AttributeValue>
        <ResourceAttributeDesignator AttirbuteId="xacml:resource:resource-id"
            DataType="xacml2:data-type:xpath-expression" />
      </ResourceMatch>
    </Resource></Resources></Target>
    <Condition><Apply FunctionId="thesis:function:string-starts-with">
      <Apply FunctionId="xacml:function:string-one-and-only">
        <AttributeSelector DataType="xs:string"
            RequestContextPath="//p:param[@param='docroot']/p:value/text()" />
      </Apply>
      <Apply FunctionId="xacml2:function:string-concatenate">
        <AttributeValue DataType="xs:string">/home/</AttributeValue>
        <Apply FunctionId="xacml:function:string-one-and-only">
          <SubjectAttributeDesignator DataType="xs:string"
            AttributeId="xacml:subject:subject-id" />
        </Apply>
      </Apply>
    </Apply></Condition>
  </Rule>
</Policy>
```

Figure 10: The XACML policy that only allows users from the webuser group to add vhosts with a document root in their homedirectory.

Our prototype is able to provide this by analysing the changes made to a configuration file and deriving the actions from it that need to be authorised. The prototype is flexible enough to do this in a very fine-grained manner. Using the ACHEL method changes to Puppet manifests are authorised at the level of instantiating resources and authorising them based on the parameters and the scope they are declared in.

Second, our prototype is not yet fully finished. It is possible to circumvent the authorisation using simple attacks. This is not a limitation of our approach but of the XACML language expressiveness which results in a policy that is not fully closed. A possible solution for this type of attacks is extending the standard XACML function-space to include domain-specific helper functions to create a fully closed access control policies. In our prototype we used the enterprise-java-xacml [12] XACML engine. Adding functions to this engine is easy as adding an annotation to a Java class and the method that implements the XACML function.

7 Future work

We added basic support for our authorisation mechanism to Puppet. Future work in this direction should focus on extending support for the Puppet language and on integrating update workflows in the authorisation phase.

Extending Puppet support Currently Puppet support is limited to a subset of the Puppet language. This subset provides a usable implementation, especially using the design patterns we described. Our prototype can be extended to support additional language constructs such as calling functions or exporting resources.

Integrating workflow In our original paper [10] we also integrated workflow support. This support is orthogonal to extracting meaningful changes from changes in Puppet manifests. Therefore we did not implement this in this prototype. This workflow support is based on digital signatures on revisions in the version repository. This information can be made available to the XACML engine through a PIP. This should be sufficient to add the workflow support to this prototype.

Ownership information Our ACHEL method can also track ownership information based on the changes made to the configuration files. With this ownership information a policy could also include that person A cannot change any parameters or resources that are owned by person B. To derive ownership information we need to start from the first revision and determine for each change what the impact is on the ownership of a statement. For Puppet one of these questions is who is the owner of a resource? Is this the user that gave the name to the resource? If parameters are changed, how does the ownership of the resource and the parameter change?

8 Conclusion

In this paper we developed a prototype that authorises changes to the Puppet input language based on their meaning. It derives the operations that need to be authorised from the changes to the input. This prototype extends our previous work by applying it to a real system configuration tool with a complex input language. These changes are authorised using XACML which is a widely adopted industry standard for access control and authorisation, instead of using regular expressions.

In our previous work we applied our approach to a simple configuration language. The results from this work show that although the configuration language is more complex, it is possible to extract the individual meaningful configuration changes. The complete language is not yet fully supported, and the prototype needs more work to analyze the difficult language constructs before it can be used in production. Our claim from our previous paper that the method is language agnostic holds, on the condition that the method can start from a clean AST. The usage of the XACML standard for authorisation provides a lot of flexibility for writing policies, as well as extensibility and integration of other information sources. This prototype does not include the workflow enforcement of our Achel [10] prototype but can be easily added to the XACML engine by including a PIP that provides the signature information required to enforce workflows.

This implementation uses XACML as authorisation language. In our first prototype we used a regular expression based language we developed. XACML provides an authorisation engine that is the de facto industry standard in contrast to our own authorisation language. The usability of our regular expression based language was also very poor. Unfortunately it appears to be hard to write policies in XACML as well. Luckily tooling support exists for writing XACML policies to improve usability.

To conclude the main contributions of this paper are: First of all the identification of the difficulties and possibilities to extract meaningful changes from a complex configuration language such as Puppet. Second the development of a set of rules that describe how to write a policy and refer to the configuration elements in these policies.

9 Acknowledgements

This research is partially funded by the Agency for Innovation by Science and Technology in Flanders (IWT), by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

[1] Puppet Website. `http://www.puppetlabs.com`, 2010.

[2] ANDERSON, P. *Short Topics in System Administration 14: System Configuration*. Berkeley, CA, 2006.

[3] BARRETT, R., MAGLIO, P. P., KANDOGAN, E., AND BAILEY, J. Usable autonomic computing systems: The system administrators' perspective. *Advanced Engineering Informatics 19*, 3 (2005), 213 – 221. Autonomic Computing.

[4] CHAWATHE, S. S., AND GARCIA-MOLINA, H. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data - SIGMOD 97 SIGMOD 97* (New York, NY, USA, 1997), ACM, pp. 26–37.

[5] DELAET, T., JOOSEN, W., AND VANBRABANT, B. A survey of system configuration tools. In *Proceedings of the 24th Large Installations Systems Administration (LISA) conference* (San Jose, CA, USA, 11/2010 2010), Usenix Association, Usenix Association.

[6] HIGGS, C. Authorisation and Delegation in the Machination Configuration System. In *Proceedings of the 22nd Large Installation System Administration (LISA) Conference* (Berkeley, CA, USA, 2008), USENIX Association, pp. 191–199.

[7] MCCARTHY, J. Towards a mathematical science of computation. *Information Processing 62* (1962), 21–28.

[8] MOSES, T. *eXtensible Access Control Markup Language (XACML) Version 2.0*, februari 2005. `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`.

[9] MYERS, E. W. An O(ND) difference algorithm and its variations. *Algorithmica 1*, 1 (1986), 251–266.

[10] VANBRABANT, B., DELAET, T., AND JOOSEN, W. Federated access control and workflow enforcement in systems configuration. In *Proceedings of the 23rd Large Installations Systems Administration (LISA) conference* (Baltimore, MD, USA, 11/2009 2009), Usenix Association, Usenix Association, p. 129–143.

[11] VELASQUEZ, N. F., AND WEISBAND, S. P. Work practices of system administrators: implications for tool design. In *CHiMiT '08: Proceedings of the 2nd ACM Symposium on Computer Human Interaction for Management of Information Technology* (New York, NY, USA, 2008), ACM, ACM, pp. 1–10.

[12] WANG, Z. Enterprise Java XACML Implementation. `http://code.google.com/p/enterprise-java-xacml/`, december 2010.