Defining and Measuring Policy Coverage in Testing Access Control Policies

Evan Martin, Tao Xie, Ting Yu
Department of Computer Science
North Carolina State University
Raleigh NC USA 27695
{eemartin,xie,yu}@csc.ncsu.edu

Abstract

To facilitate managing access control in a system, security officers increasingly write access control policies in specification languages such as XACML, and use a dedicated software component called Policy Decision Point (PDP). To increase confidence on written policies, certain types of policy testing (often in an ad hoc way) are usually conducted, which probe PDP with some typical requests and check PDP's responses against expected ones.

This paper develops a first step toward systematic policy testing by defining and measuring policy coverage when testing policies. We have developed a coverage-measurement tool to measure policy coverage given a set of XACML policies and a set of requests. We have developed a tool for request generation, which randomly generates requests for a given set of policies, and a tool for request reduction, which greedily selects a nearly minimal set of requests for achieving the same coverage as the originally generated requests. To evaluate coverage-based request reduction and its effect on fault detection, we have conducted an experiment with mutation testing on a set of real policies. Our experimental results show that the coverage-based test reduction can substantially reduce the size of generated requests and incur only relatively low loss on fault detection. We also conduct a study on the policy coverage achieved by manually generated requests.

1 Introduction

Access control is one of the most fundamental and widely used security mechanisms. It controls which principals (users, processes, etc.) have access to which resources in a system. To better manage access control, systems often explicitly specify access control policies using policy languages such as XACML [1] and Ponder [8]. Whenever a principal requests access to a resource, that request is passed to a software component called Policy Decision Point (PDP). PDP evaluates the request against access control policies, and grants or denies the request accordingly.

The specification of access control policies is often a challenging problem. It is common that a system's security is compromised due to the misconfiguration of access control policies instead of the failure of cryptographic primitives or protocols. This problem becomes increasingly severe as software systems become more and more complex, and are deployed to manage a large amount of sensitive information and resources that are organized into sophisticated structures.

Formal verification is an important means to ensuring the correct specification of access control policies. Recently, several tools have been developed to verify XACML access control policies against user-specified properties [10, 13, 24]. However, it is often beyond the capabilities of these tools to verify complex access control policies in large-scale information systems. Further, user-specified properties are often not available [10].

Like in software development, errors in access control policies may also be discovered through testing. In fact, once access control policies are specified, they are often tested with some access requests so that security officers may check the PDP's responses against expected ones [5]. However, current policy testing practice tends to be ad hoc. Although there exist various coverage criteria [25] for software programs, there are no criteria or good heuristics to guide systematic generation of high-quality policy test suites. With an ad hoc policy testing, it is questionable that high confidence could be gained on the correctness of access control policies.

This paper presents a first step toward systematic policy testing. We propose the concept of *policy coverage* to measure the quality of policy test suites, which are a set of request-response pairs. Intuitively, the more policy rules (as well as their components such as subjects, resources, and conditions) are involved when evaluating a test suite, the more likely it is to discover errors in access control poli-

cies. We have developed a coverage-measurement tool to measure the coverage of XACML policies achieved by a set of access requests. We have also developed a request-generation tool that randomly generates a policy test suites for a given set of policies.

Though the randomly generated test suites can achieve high policy coverage, and are effective in detecting a variety of policy specification errors, it may potentially include a huge number of requests, which makes it difficult to efficiently inspect and verify the correctness of responses from the PDP. To mitigate this problem, we further propose a request reduction technique to significantly reduce the size of a test suite while maintaining its policy coverage.

Previous experiments [20] showed that test reduction based on program code coverage can severely compromise the fault-detection capabilities of the original test suite. To evaluate the impact of the proposed request reduction technique on the quality of policy testing, we conduct an experiment on a set of real policies with mutation testing [9], which is a specific form of fault injection that consists of creating faulty versions of a policy by making small syntactic changes. In the experiment, we compare the fault-detection capabilities of the reduced set and original set of requests. Our experimental results show that our coverage-based request reduction technique can substantially reduce the size of generated requests but incur only relatively low loss in fault detection capabilities. We also conduct a study by measuring the policy coverage of an XACML conformance test suite and requests for a conference reviewing system's policy. Our results show that the measurement of policy coverage can effectively identify uncovered parts of policies. Such results can be used to guide the development of further test cases, significantly improving the quality of policy testing.

The rest of the paper is organized as follows. Section 2 discusses related work and background information. Section 3 proposes the concept of policy testing and policy coverage based on a general access control model. In Section 4, we instantiate the concept of policy coverage in the context of XACML, a widely used and standardized meta policy language for expressing domain-specific access control requirements. We also presents the design of a coverage measurement tool. Sections 5 and 6 describe the request-generation tool and our request reduction technique, respectively. Section 7 presents a set of initial mutation operators developed for policies. Section 8 presents the experiment conducted to assess request reduction and its effect on fault detection capabilities. Section 9 illustrates the study of measuring the policy coverage achieved by manually generated requests. Section 10 concludes the paper with future directions.

2 Related Work and Background

Several tools have been developed to verify properties for XACML policies [1]. Hughes and Bultan translated XACML policies to the Alloy language [14] and check their properties using the Alloy Analyzer. Fisler et al. [10] developed a tool called Margrave that uses multi-terminal binary decision diagrams [7] to verify user-specified properties and perform change-impact analysis. Zhang et al [24] developed a model-checking algorithm and tool support to evaluate access control policies written in *RW* languages, which can be converted to XACML [23]. These existing approaches assume that policies are specified using a simplified version of XACML. It is challenging to generalize these verification approaches to support full-feature XACML policies with complex conditions. In addition, most of these approaches require users to specify a set of properties to be verified; however, policy properties often do not exist in practice. The systematic policy testing approach proposed in this paper works on full-feature XACML policies without requiring properties, complementing the existing policy verification approaches.

A test adequacy or coverage criterion provides a stopping rule for testing and a measurement of a test suite's quality [25]. A test coverage criterion can be used to guide test selection. A coverage criterion typically specifies testing requirements based on whether all the identified features in a program or specification have been fully exercised. Identified features in a program can be statements, branches, paths, or definition-use paths. Identified features in a specification can be choices for categories [3,4] or conditions [6] in specifications.

The importance of test coverage criterion in fault detection can be shown through a fault propagation model such as the PIE (Propagation, Infection, and Execution) model [22]. For example, in order to expose a bug in a statement in a program, a test needs to at least cover the buggy statement. Note that the coverage of a buggy statement is not a sufficient condition to expose the buggy behavior in program outputs; additionally the execution of the buggy statement needs to produce a wrong data state and the wrong data state needs to have an effect on program outputs.

With our knowledge, our approach is the first that proposes policy coverage and develops an automatic measurement tool and a request reduction tool for it. But there exist several approaches for defining and measuring coverage of rules for grammar-based software or SQL statements for database applications. For example, Hennessy and Power [12] defined rule coverage for context-free grammar and used rule coverage to reduce a test suite for grammar-based software such as C++ compilers. Suarez-Cabal and Tuya [21] defined coverage of SQL queries and developed a tool to automate the measurement. Kapfhammer and Soffa [16] defined a family of test adequacy criteria for

database-driven applications based on dataflow information that is associated with entities in a database. Different from these existing coverage measurement approaches for grammars, SQL queries, or database entities, our new approach defines and measures coverage information for policies.

3 Access Control Policies and Policy Coverage

Many access control policy languages have been proposed for different application domains. Policies in these languages are usually composed of a set of rules which specify under what conditions a subject is allowed or denied access to certain objects in a system. To discuss policy coverage criteria in general, we model access requests and policies in this paper as follows.

Let $\mathcal{S}$, $\mathcal{O}$ and $\mathcal{A}$ denote respectively the set of all the subjects, objects and actions in an access control system. Each subject, object, or action is associated with a set of attributes that may be used for access control decisions. For example, a subject's attributes may include a user's role, rank and security clearance. An object's attributes may include a file's type, a document's security class, and a printer's location.

An access request q is a tuple (s, o, a), where $s \in \mathcal{S}$, $o \in \mathcal{O}$ and $a \in \mathcal{A}$. A request (s, o, a) means that subject s requests to take action a on object o.

An access control policy P is a sequence of rules, each of which is of the form $(Cond_s, Cond_o, Cond_a, decision, Cond_g)$. $Cond_s$, $Cond_o$ and $Cond_a$ are constraints over the attributes of a subject, object and action respectively. $Cond_g$ is a general constraint which may potentially be over all the attributes of subjects, objects, actions and other properties of a system (e.g., the current time and the load of a system). A $decision$ is either $deny$ or $permit$. Given a request (s, o, a), if $Cond_s(s)$, $Cond_o(o)$, $Cond_a(a)$ and $Cond_g$ are all evaluated to be $true$, then the request is either permitted or denied, according to $decision$ in the rule.

One may wonder that since $Cond_g$ can be a general constraint over the attributes of subjects, objects, actions as well as other properties of a system, why we still need $Cond_s$, $Cond_o$ and $Cond_a$ in a rule? The reason is that, though conceptually those conditions can be merged with the general condition $Cond_g$, by separating them, it makes it easy to quickly locate relevant rules to a request. For example, given a request (s, o, a), if one of $Cond_s$, $Cond_o$ and $Cond_a$ is evaluated to be false, then we do not need to further evaluate $Cond_g$ that sometimes may be much more complex than the former three. Such a form of access control rules is commonly supported in access control policy languages. If a request satisfies $Cond_s$, $Cond_o$ and $Cond_a$ of a rule, then we say the rule is *applicable* to the request.

A policy may have multiple rules that are applicable to a request. These rules may in fact offer conflicting decisions. The final decision regarding the request depends on application specific conflict resolution functions. Commonly used conflict resolution functions include denial overriding permission (where a request is denied if it is denied by at least one rule), permission overriding denial (where a request is permitted if it is permitted by at least one rule) and first applicable (where the final decision is the same as that of the first applicable rule in a sequence of rules whose condition $Cond_g$ is evaluated true). We use PDP (Policy Decision Point) to denote the component of a system where final decisions are made according to the decision of each rule and a specific confliction resolution function. Conceptually, given a policy P and a request q, a PDP returns the access control decision of q.

Since we are interested in capturing potential errors in policy specifications, we assume that PDP is correctly implemented in the rest of the paper. In practice, generic PDP implementations are often available which has been scrutinized by the public.

We next start our discussion on policy testing based on the above model. The basic idea of policy testing is very simple. Like software testing, given a policy, we would like to generate a set of requests, and check whether the access control decisions on these requests are as expected. Any unexpected decision indicates potential errors in the specification of the policy.

Clearly, if no requests are evaluated against a rule during testing, then potential errors in that rule will not be discovered. Thus, it is important to generate requests so that a large portion of rules are involved in the evaluation of at least one of the requests. In other words, we are interested in requests that cause a rule's conditions to be evaluated to be true.

Definition 1 *Given a request q and a rule m in a policy P, we say q* covers *m if m is applicable to q. Given a set of requests $\mathcal{Q}$, the rule coverage of P by $\mathcal{Q}$ is the ratio between the number of rules covered by at least one request in $\mathcal{Q}$ and the total number of rules in P.*

Intuitively, the higher the rule coverage of a set of request, the better chance specification errors may be discovered. Like software testing, it is often infeasible to have exhausted policy testing when the space of possible requests is large. Therefore, policy specification errors may still exist even after testing with requests that cover all the rules.

To improve the quality of policy testing, it helps to further examine potential errors in the specification of conditions in each rule, which can also be tested by requests.

Definition 2 *Given a request q and a rule $m(Cond_s, Cond_o, Cond_a, decision, Cond_g)$, we say $Cond_g$ is* positively (negatively) covered *by q if m is covered by q and $Cond_g$ is evaluated to be true (false). Given a set of requests $\mathcal{Q}$, the condition coverage of P by*

$\mathcal{Q}$ is the ratio between the numbers of general conditions positively or negatively covered by at least one request in $\mathcal{Q}$ and two times of the total number of rules in P.

The intuition behinds the above definition is as follows. An error in the condition of a rule may have two types of impacts on a request. Suppose $Cond'_g$ is the condition when an error is introduced to the original condition $Cond_g$. Given a request q, $Cond'_g(q)$ may be evaluated to be true while $Cond_g(q)$ is false, or vice versa. That is why we concern with both positive and negative coverage of a condition in the above definition.

4 Policy Testing in XACML

In this section, we focus our discussion of policy testing on XACML (eXtensible Access Control Markup Language). XACML is a language specification standard designed by OASIS. It can be used to express domain-specific access control policy languages as well as access request languages. Besides offering a large set of built-in functions, data types and combining logic, XACML also provides standard extension interfaces for defining application specific features. Since it was proposed, XACML has received much attention from both the academia and the industry. Many domain specific access control languages have been developed using XACML [17, 19]. Open source XACML implementations are also be provided for different platforms (e.g., Sun's XACML implementation and XACML.NET). Therefore, XACML provides an ideal platform for the development of policy testing techniques so that they can be easily applied to multiple domains and applications. In this section, we first give a brief introduction of XACML, and then discuss the policy coverage criteria for XACML.

4.1 XACML

The basic concepts of access control in XACML include *policies*, *rules*, *targets* and *conditions*. A single access control policy is represented by a policy element, which includes a target element and one or more rule elements. A target element contains a set of constraints on the subject (e.g., the subject's role is equal to faculty), resources (e.g., the resource name is grades), and actions (e.g., the action name is assign)[1].

A target specifies to what kinds of requests a policy can be applied. If a request cannot satisfy the constraints in the target, then the whole policy element can be skipped without further examining its rules.

We next describe how a policy is applied to a request in details. A policy element contains a sequence of rule elements. Each rule also has its own target, which is used to determine whether the rule is applicable to a request. If a rule is applicable, a *condition* (a boolean function) associated with the rule is evaluated. If the condition is evaluated to be true, the rule's *effect* (`Permit` or `Deny`) is returned as a *decision*; otherwise, `NotApplicable` is returned as a decision. If an error occurs when a request is applied against policies or their rules, `Indeterminate` is returned as a decision.

More than one rule in a policy may be applicable to a given request. To resolve conflicting decisions from different rules, a *rule combining algorithm* can be specified to combine multiple rule decisions into a single decision. For example, a deny overrides algorithm determines to return `Deny` if any rule evaluation returns `Deny` or no rule is applicable. A first applicable algorithm determines to return what the evaluation of the first applicable rule returns.

In general, an XACML policy specification may also include multiple policies, which are included with a container element called *PolicySet*. When a request can also be applied to multiple policies, a *policy combining algorithm* can also be specified in a similar way.

Figure 1 shows an example XACML policy, which is revised and simplified from a sample Fedora[2] policy (to be

```
<Policy PolicyId="demo" RuleCombinationAlgId="first-applicable">
 <Target>
   <Subjects> <AnySubjects/> </Subjects>
   <Resources>
    <Resource>
     <ResourceMatch MatchId="equal">
      <AttributeValue>demo:5</AttributeValue>
      <ResourceAttributeDesignator AttributeId="objectid"/>
     </ResourceMatch>
    </Resource>
   </Resources>
   <Actions> <AnyAction/></Actions>
 </Target>
 <Rule RuleId="1" Effect="Deny">
  <Target> <Subjects><AnySubject/></Subjects>
   <Resources> <AnyResource/> </Resources>
   <Actions>
    <Action>
     <ActionMatch MatchId="equal">
      <AttributeValue>Dissemination</AttributeValue>
      <ActionAttributeDesignator AttributeId="actionid"/>
     </ActionMatch>
    </Action>
   </Actions>
  </Target>
  <Condition FunctionId="not">
   <Apply FunctionId="at-least-one-member-of">
    <SubjectAttributeDesignator AttributeId="loginid"/>
    <Apply FunctionId="string-bag">
     <AttributeValue>testuser1</AttributeValue>
     <AttributeValue>testuser2</AttributeValue>
     <AttributeValue>fedoraAdmin</AttributeValue>
    </Apply>
   </Apply>
  </Condition>
 </Rule>
 <Rule RuleId="2" Effect="Permit"/>
</policy>
```

Figure 1. An example XACML policy

[1] Conditions of "AnySubject", "AnyResource", and "AnyAction" can be satisfied by any subject, resource, or action, respectively.

[2] `http://www.fedora.info`

used in our experiment described in Section 8). This policy has one policy element which in turn contains two rules. The rule composition function is "first-applicable", whose meaning has been explained before. Line 2-13 defines the target of the policy, which indicates that this policy only applies to those access requests of an object "demo:5". The target of rule 1 (line 15-25) further narrows the scope of applicable requests to those asking to perform "Dissemination" action on object "demo:5". Its condition (line 26-35) indicates that if the subject's "loginId" is "testuser1", "testuser2", or "fedoraAdmin", then the request should be denied. Otherwise, according to rule 2 (line 37) and the rule composition function of the policy (line 1), a request applicable to the policy should be permitted.

4.2 Policy Coverage in XACML

In XACML languages, we can see there are three major entities: policies, rules for each policy, and a condition for each rule. We define policy coverage as follows:

- *Policy hit percentage*. A policy is hit by a request if the policy is applicable to the request; in other words, all the conditions in the policy's target are satisfied by the request. Policy hit percentage is the number of hit policies divided by the number of total policies.
- *Rule hit percentage.* A rule for a policy is hit by a request if the rule is also applicable to the request; in other words, the policy is applicable to the request and all the conditions in the the rule's target are satisfied by the request. Rule hit percentage is the number of hit rules divided by the number of total rules.
- *Condition hit percentage.* The evaluation of the condition for a rule has two outcomes: true and false, which are called as the true condition and false condition, respectively. A true condition for a rule is hit by a request if the rule is applicable to the request and the condition is evaluated to be true. A false condition for a rule is hit by a request if the rule is applicable to the request and the condition is evaluated to be false. Condition hit percentage is the number of hit true conditions and hit false conditions divided by twice of the number of total conditions.

Note that a policy has at least one rule but a rule can have no condition, indicating an implicit condition `true`, which is always satisfied when the rule is applicable. Therefore, when there are no conditions defined within the policies under consideration, the condition hit percentage is always the same as the rule hit percentage. Normally a policy tester shall be able to generate requests to achieve 100% for all three types of policy coverage. In other words, all the to-be-covered entities defined in the policy coverage are feasible to cover in principle; otherwise, those infeasible parts of policy specifications could be removed like dead code in programs.

To automate the measurement of policy coverage, we have developed a measurement tool based on Sun's open source XACML implementation [2], written in Java. Based on Sun's XACML implementation, we first built a Policy Decision Point (PDP), which receives an access request and returns an access decision. We then developed several public methods in a Java class for collecting runtime coverage and insert some call sites to these methods in several places in the code of Sun's XACML implementation. When PDP loads given policies, we insert a method call to collect all the policies, rules, and conditions in the given policies. Every time PDP determines that all the conditions in a policy's target are satisfied, we insert a method call to collect policy hit information. Every time PDP determines that all the conditions in a rule's target are satisfied, we insert a method call to collect rule hit information. Every time PDP determines that a condition for a rule is evaluated to be true or false, we insert a method call to collect condition hit information.

After PDP returns the decision, we output the coverage information into a text file, whose name is determined by the names of given policies; if a text file with the same name exists, the coverage information in it is updated by incorporating the new coverage information. Therefore, when PDP receives several requests separately against the same set of policies, the aggregated coverage information achieved by them is collected. Besides the basic coverage information, we also output the details of covered entities and their covering requests as well as the details of uncovered entities. The extra information can help developers or external tools in generating or selecting requests for achieving higher policy coverage.

5 Request Generation

To generate requests automatically for achieving policy coverage, we have developed a request-generation tool for inspecting the policy under test and constructing a request factory that provides requests on demand. There are various algorithms that can be devised to generate requests. At present we have implemented two simple factories called the `AllComboReqFactory` and the `RandomReqFactory`. The former attempts to generate requests for all possible combinations of attribute id-value pairs found in the policy while the latter randomly selects requests from the set of all combinations. This is achieved by representing a particular request as a vector of bits. The length of this vector is equal to the number of different attribute values found in the policy targets, rule targets, and rule conditions of the policy under test. Each attribute value appears in the request if its corresponding bit in the vector is 1, otherwise the value is not present.

To generate all possible combinations we increment an

integer i from 0 to 2^n where n is the number of attribute values found in the policy. To construct a request from the integer i we first convert i to binary and use the n least significant bits as the vector of bits that indicate the presence or absence of the possible attribute values. This approach guarantees that all possible combinations of the available attribute values are generated. However this is a simplistic approach and not realistic for larger policies since the number of possible requests increases exponentially with the number of possible attribute values. In addition to this shortcoming, there are instances in which the set of attribute values is not finite, such as cases with integer data types and greater than or less than condition functions. Such instances make the use of the `AllComboReqFactory` impractical or even impossible.

The generation of random requests is done in a similar fashion. First, the policy is inspected and the n possible attribute values are determined. Each request is generated by setting each bit in the vector to 0 or 1 with probability 0.5. The number of randomly generated requests can be configured by the user and the configured number can be considerably smaller than the total number of combinations. To help achieve adequate coverage with a small set of random requests, we further modified this algorithm to ensure that each bit was set to 1 and 0 at least once. We accomplish this by explicitly setting the i^{th} bit to 1 for the first n generated requests where $i = 1, 2, ...n$. Similarly, for the next n requests, we explicitly set the $(i - n)^{th}$ bit to 0. This approach guarantees that each attribute value is present and absent at least once as long as the number of randomly generated requests is greater than $2n$.

6 Request Reduction

The request reduction problem can be stated similar to the test minimization problem for program testing [11]:

Given: request set QS, a set of requirements r_1, r_2, ..., r_n that must be satisfied to provide the desired test coverage of the policies, and subsets of QS, Q_1, Q_2,..., Q_n, one associated with each of the r_is such that any one of the request q_j belonging to Q_i can be used to test r_i.
Problem: Find a representative set of requests from QS that satisfies all of r_is.

In the problem statement, the r_is can represent policy coverage requirements, such as covering a certain policy, a certain rule, and a certain condition. In a representative set of requests that satisfies all of the r_is, at least one request satisfies each r_i. We call a representative set is *minimal* if removing any request from the set causes the set not to be a representative set. Given a request set QS, there can be several minimal representative sets $QS' \subseteq QS$. Among the minimal representative request sets, we could find a request set that has the smallest possible number of requests. Finding such request tests reduces to optimization problems called "minimum set cover" and "minimum exact cover", respectively; these problems are known to be NP complete, and in practice approximation algorithms are used [15].

In our implementation of coverage-based request reduction, we use a greedy algorithm for selecting requests as they are generated by the random request factory if and only if the generated request increases any of the coverage metrics described in Section 4. More specifically, we iteratively generate a random request, add it to the large set. We then evaluate that request against the policy in order to both compute the response and measure the coverage. If the coverage increases due to the evaluation of the request, then that request is added to the reduced request set.

We note that this greedy algorithm may not produce a minimal representative set. In practice, it does, however, often produce a representative set whose size is near the size of a minimal representative set. We call our reduced set as a *nearly minimal* representative set.

7 Mutation Testing

In order to investigate the effect of request reduction on fault detection capabilities, we can inject faults into the original policy thereby creating faulty policies. Since fault detection is the central focus of any testing process, it provides an external measure of the effectiveness of that process. We aim to demonstrate that reduced request sets based on coverage can detect a large percentage of the faults detected by the original request set. We use mutation testing [9] as a mechanism to compare request sets in terms of fault detection. In mutation testing, the policy under test is mutated to introduce an error and a request set is evaluated against the mutant policy. If the request set produces any response that differs from the corresponding response produced by the original policy, then the mutant is killed.

There are many studies concerned with the types and effectiveness of mutating general purpose programming languages [12, 18], however many of these do not directly apply to mutating policies. We describe the types of mutation operators employed in our experiments in Table 1.

8 Experiment on Request Reduction and Its Effect on Fault Detection

The objective of the experiment is to examine whether the reduced request set will be as effective at fault detection as the original request set. Similar to the goals of Hennessy et al. [12] for grammar-based software, we wish to investigate the following hypotheses:

Hypothesis 1 *We can achieve a significant reduction in request-set size for large randomly generated request sets*

Table 1. The types of mutation operator applied to the policies.

ID	Description
REff	Invert the Rule `Effect` by changing `Permit` to `Deny` or `Deny` to `Permit`.
PTargT	Ensure the policy is always applied to all requests by replacing tags within the `<Target>` tag with `<Any*/>`.
PTargF	Ensure the policy is never applied to any requests by removing tags within the `<Target>`.
RTargT	Ensure the rule is always applied to all requests by replacing tags within the `<Target>` tag with `<Any*/>`.
RTargF	Ensure the rule is never applied to any requests by removing tags within the `<Target>`.
CondT	Ensure the condition always evaluates to True by removing the condition entirely.
CondF	Ensure the condition always evaluates to `False` by manipulating the condition value or the condition function.
POrder	Try all combinations of policy orderings. This mutant is only meaningful if there is more than one policy and the policy combining algorithm is order sensitive.
ROrder	Try all combinations of rule orderings. This mutant is only meaningful if there is more than one rule and the rule combining algorithm is order sensitive.

while maintaining equivalent policy, rule, and condition coverage.

Hypothesis 2 *Reducing a request set based on coverage will not proportionately decrease its fault detection capability.*

In order to investigate our hypotheses, we need to measure the reduction in request-set size, the coverage metrics, and the reduction in fault detection capability. We measure each of these against three XACML policies used by Fedora[3], an open source software that gives organizations a flexible service-oriented architecture for managing and delivering digital content. Fedora leverages Sun's XACML implementation [2] to provide fine-grained access control to the digital content it manages. The Fedora repository of default and example XACML policies proved a useful resource for realistic test subjects. The subjects selected for this experiment are apia-tighten, demo-5, and the default set of policies shipped with Fedora. The basic metrics of each policy include the number of policies, policy targets, rules, rule targets, and conditions found in each of the three subjects. These metrics are summarized in Table 2. The first policy, apia-tighten, is just one of a set of policies used to tighten the access control of the Fedora system. The overall intent of the policy is datastream hiding, meaning that raw datastreams must not be accessible to anyone except very privileged users. The demo-5 policy is an example policy used to show how access control can be enforced on particular objects. The default policy is actually a set of 13 policies that approximate the access control of an earlier version of Fedora.

We first randomly generate requests for each policy as outlined in Section 5 and greedily select a smaller set of requests with equivalent coverage as outlined in Section 6. If we define the size of the entire request set as r and the size of the reduced request set as r' then we can define the

[3] `http://www.fedora.info`

Table 2. Basic policy metrics of the experimental subjects.

Policy	apia-tighten	demo-5	default
# Policies	1	1	13
# Policy Targets	1	1	13
# Rules	2	3	13
# Rule Targets	2	2	0
# Conditions	2	2	6

reduction in request-set size, $SizeReduction$, as follows:

$$SizeReduction = 1 - \frac{r'}{r}$$

Table 3 shows the size of the generated request set, the size of the reduced request set, the coverage obtained by each request set, and the reduction in request-set size achieved by the greedy selection algorithm. The results in Table 3 show that we can achieve more than 98% size reduction for the three Fedora policies. The results suggests that we can indeed greatly reduce the request set size of large randomly generated request sets while maintaining equivalent policy, rule, and condition coverage.

The second objective of the experiment is to investigate if the reduced request set can still effectively detect faults in policies compared to the full set. We perform the experiment illustrated in Figure 2. The basic approach is to exploit mutation testing as a mechanism to compare the fault detection capability of various request sets. As discussed in Section 7, we create several mutant policies using the mutation operators listed in Table 1 for each of the experimental subjects. The quantity and type of mutation operator used for each policy is summarized in Table 4.

Each request set is executed against each mutant policy and their corresponding responses are recorded. If the response for any request evaluated against the original policy

Table 3. Coverage metrics and reduction in request-set size achieved for each policy.

Policy	apia-tighten	demo-5	default
Coverage Metrics			
Policy Hit %	100%	100%	100%
Rule Hit %	100%	100%	100%
Condition Hit %	100%	100%	91.67%
Request Reduction			
# Requests	200	200	1000
# Reduced Requests	4	3	10
Size Reduction	98%	98.5%	99%

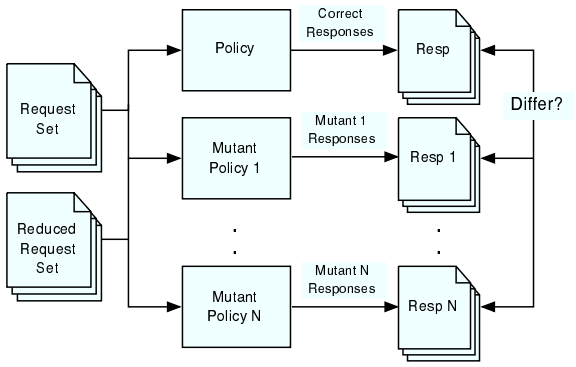

Figure 2. Overview of fault detection experiment.

Table 4. Type of mutation operator and quantity of mutant policies created for each policy.

Policy	apia-tighten	demo-5	default
Mutation Operator Id			
REff	2	3	13
PTargT	1	1	13
PTargF	1	1	13
RTargT	2	2	0
RTargF	2	2	0
CondT	2	2	6
CondF	2	2	6
POrder	0	0	0
ROrder	1	5	
Total Mutants	13	18	51

differs from the response for the request evaluated against the mutant policy, then the mutant is said to be killed. We define the $CapabilityReduction$ as a metric that quantifies the relative fault detection capability of the reduced set compared to its original set. If we define the total number of mutants detected by the original set as m and the total number of mutants detected by the reduced set as m', then we compute the reduction in fault detection as:

$$CapabilityReduction = 1 - \frac{m'}{m}$$

Table 5 summarizes the results of our experiments on the reduction of fault detection. The results suggest that, on average, a 98.5% reduction in request-set size only results in a 18.97% reduction in fault detection capability. In many cases we suspect the reduction in fault detection capability is acceptable considering the large reduction in request-set size. These results suggest that request sets requiring manual inspection can be greatly reduced with relatively low loss to fault detection capability.

Table 5. Reduction in fault detection.

Policy	apia-tighten	demo-5	default
# Mutants	13	18	51
Kill %			
Original Set	76.92%	94.44%	80.39%
Reduced Set	69.23%	77.78%	56.86%
Capability Reduction	10%	17.65%	29.27%

One possible explanation as to why the reduced set suffers from a reduction in fault detection capability is greedy selection of requests for addition to the reduced set. Since requests can carry multiple attribute values for the same attribute ids, it is possible that two requests with the identical coverage can produce different responses by operating on the same conditions in different ways. There may be more optimal, albeit likely more complex, algorithms for choosing requests.

We consider these results promising yet further experimentation with larger, more complex policies and a more comprehensive suite of mutation operators is necessary to further validate the findings. We will investigate if there are any specific types of mutation operators that result in mutants that are least likely to be killed by the reduced set.

9 Empirical Study of Manually Generated Requests' Policy Coverage

We have applied the coverage-measurement tool on the whole set of the XACML committee specification conformance test suite [5] and a conference paper review system's policy and its requests developed by Zhang et al. [23].

Table 6. Policy coverage of the XACML conformance test suite

type	100% all	50% cond	non-0% rule/cond	0% rule/cond	total
policies	24	172	24	14	234
Permit	31	144	6		181
Deny			6		6
NotApp	13	28	6	10	57
Indet	1	2	6	4	13

The XACML conformance test suite includes 337 distinct policies[4], 374 requests, their expected responses from the application of the policies. Among these 337 distinct policies, we show the results of 234 policies in this section because for the requests of the remaining 103 policies, Sun's XACML implementation [2] responded different decisions than the ones specified in their expected responses. Applying the requests on these 103 policies failed to conform with expected responses because Sun's XACML implementation does not support some optional features of XACML specifications.

The conference paper review system's policy specified by Zhang et al. [23] has 11 requests and 15 rules, which have 10 conditions. These 10 conditions involve the execution of SQL statements that access an external database. Because it is not trivial to adapt Sun's XACML implementation to support this, we simply remove these 10 conditions as well as some attributes that are not parsed by Sun's XACML implementation, in order to allow us to focus on the the measurement of rule hit percentage.

After we fed 374 requests in the XACML conformance test suite to the coverage-measurement tool, we summarize the reported statistics of policy coverage in Table 6. Note that all policies in the conformance test suite are hit by the requests, achieving 100% policy hit percentage. Column 1 shows the type of data and Columns 2-5 show the data for different types of coverage. Row 2 shows the number of policies. Rows 3-6 show the number of requests whose returned decisions are `Permit`, `Deny`, `NotApplicable`, and `Indeterminate`, respectively. When a data entry has a zero value, we do not show the zero value but leave the entry empty.

Column 2 shows the data for policies whose policy, rule, and condition hit percentages reach 100%. These policies have achieved the optimal policy coverage. Column 3 shows the data for policies whose policy and rule hit percentages reach 100% but condition hit percentage reaches 50%. These policies achieve almost-optimal policy coverage because sometimes it is not very essential to cause a condition of a rule to be evaluated to be false. Column 4 shows the data for policies whose rule or condition hit percentage is less than 100% but not equal to 0% (but we do not include the cases shown in Column 3 here). The coverage of these policies needs to be improved. Column 5 shows the data for policies whose rule or condition hit percentage is equal to 0%. These policies are especially in need for improvement. The last column shows the sum of all the data in Columns 2-5.

From the results shown in Table 6, we observed that a majority of policies fell into the category of Column 3, where policy and rule hit percentages reach 100% but condition hit percentage reaches 50%. Many polices in the XACML conformance test suite contain single rules each of which has a condition. Often each of these policies receives only one request, which basically cover the policy's rule and the rule's true condition.

We took a close look at the details of 14 policies in Column 5. Two of them had 100% for rule hit coverage but 0% for condition hit percentage. Their coverage results were against our expectation because if their conditions were applicable, we expected at either true or false condition would be hit. We inspected their requests and found that a subjects's age was specified twice and their conditions access the the subject's age. When evaluating the conditions, PDP encountered an error and returned a decision of `Indeterminate`; therefore, neither true or false condition is hit.

Note that the XACML conformance test suite was not specifically constructed to achieve high coverage of policies but the measurement results still give us some insights of the common coverage distribution, reflecting policy portions that are commonly hit by manually created requests.

After we fed to coverage-measurement tool 11 requests for the conference paper review system's policy [23], 73% rule hit percentage was achieved: 4 out of 15 rules were not hit. These four uncovered rules included the case of permitting a PC chair to read papers and no request matched this case. Interestingly one of these uncovered four rules was the last rule, which has the effect of `Deny` and this rule's target can be matched by any request. This rule is often used for the `permit-overrides` rule combination algorithm [1]. Given the measurement results of coverage-measurement tool, we could construct new requests without much difficulty to cover these uncovered rules in the policy of the conference paper review system as well as those uncovered rules or conditions in many policies of the XACML conformance test suite.

10 Conclusion

In this paper, we have developed a first step toward systematic policy testing by defining and measuring policy coverage. We have proposed the concept of policy test-

[4] In the XACML conformance test suite, there are 374 policies, each of which receives a single request. We have reduced those policies with the same policy content into a single policy, which can then receive multiple requests.

ing and policy coverage based on a general access control model. We further defined three levels of specific policy coverage for XACML policies: policy hit percentage, rule hit percentage, and condition hit percentage. To support systematic policy testing based on policy coverage automatically, we have developed a coverage-measurement tool, a request-generation tool, and a request-reduction tool. By using mutation testing, we have conducted an experiment that assesses the coverage-based request reduction and its effect on fault-detection capabilities. The experimental results showed that the coverage-based request reduction substantially reduce the size of the request set but incur only relatively low loss of fault-detection capabilities. We also conducted a study on the policy coverage achieved by manually generated requests for policies in a conformance test suite for XACML specifications [5] and a conference reviewing system [23]. Our results showed that our measurement results can pinpoint uncovered areas of policies and guide the development of new requests to achieve higher policy coverage.

In future work, we plan to develop a comprehensive suite of techniques and tools for systematic policy testing. In particular, we plan to extend our policy coverage to consider cases that reflect the interactions of different rules or different policies, which are not focused by our existing policy coverage. We also plan to conduct experiments on a larger scope of policies.

References

[1] OASIS eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/xacml/, 2005.

[2] Sun's XACML implementation. http://sunxacml.sourceforge.net/, 2005.

[3] N. Amla and P. Ammann. Using Z specifications in category partition testing. In *Proc. 7th Annual Conference on Computer Assurance*, pages 3–10, June 1992.

[4] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Proc. 9th Annual Conference on Computer Assurance*, pages 69–80, June 1994.

[5] A. Anderson. XACML 1.1 committee specification conformance tests. http://www.oasis-open.org/committees/xacml/ConformanceTests/, 2002.

[6] J. Chang and D. J. Richardson. Structural specification-based testing: automated support and experimental evaluation. In *Proc. 7th ESEC/FSE*, pages 285–302, 1999.

[7] E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. International Workshop on Logic Synthesis*, 1993.

[8] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proc. International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, 2001.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[10] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proc. 27th International Conference on Software Engineering*, pages 196–205, 2005.

[11] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[12] M. Hennessy and J. F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, November 2005.

[13] G. Hughes and T. Bultan. Automated verification of access control policies. Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara, 2004.

[14] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. 8th ESEC/FSE*, pages 62–73, 2001.

[15] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974.

[16] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proc. 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–107, 2003.

[17] M. Lorch, D. Kafura, and S. Shah. An xacml-based policy management and authorization service for globus resources. In *International Workshop on Grid Computing (GRID)*, pages 208–212, Phoenix, AZ, Nov. 2003.

[18] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In *Proc. International Symposium on Software Reliability Engineering*, pages 352–363, 2002.

[19] T. Moses, A. Anderson, S. Proctor, and S. Godik. XACML Profile for Web-Services (WSPL). OASIS Working Draft, Sept. 2003.

[20] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proc. International Conference on Software Maintenance*, pages 34–43, 1998.

[21] M. J. Suarez-Cabal and J. Tuya. Using an sql coverage measurement for testing database applications. In *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2004.

[22] J. M. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, 1992.

[23] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems in XACML. In *Proc. 2004 ACM workshop on Formal Methods in Security Engineering*, pages 56–65, 2004.

[24] N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In *Proc. 8th International Conference on Information Security*, pages 446–460, September 2005.

[25] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.