Defining and Measuring Policy Coverage in Testing Access Control Policies

Evan Martin, Tao Xie, and Ting Yu

Department of Computer Science
North Carolina State University
Raleigh, NC 27695
`eemartin@ncsu.edu`,
`{xie,yu}@csc.ncsu.edu`
`http://ase.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 a 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 the 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 [14]. Whenever a principal requests access to a resource, that request is passed to a software component called a 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 [16, 22, 42]. However, it is often beyond the capabilities of these tools to verify complex access control policies in large-scale information systems. Furthermore, user-specified properties are often not available [16].

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 manually check the PDP's responses against expected ones [6]. However, current policy testing practice tends to be ad hoc. Although there exist various coverage criteria [43] 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 sets 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 policies. 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 policy test suites for a given set of policies.

Although 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 [35] 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 [15], 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 that measures the policy coverage of an XACML conformance test suite as well as 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 presents background information on XACML, a widely used and standardized meta policy language for expressing domain-specific access control requirements. 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. We also present 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 discusses related work and Section 11 concludes the paper with future directions.

2 XACML

XACML (eXtensible Access Control Markup Language) 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 [30, 32]. Open source XACML implementations are also available for different platforms (e.g., Sun's XACML implementation [2] and XACML.NET [3]). Therefore, XACML provides an ideal platform for the development of policy testing techniques that can be easily applied to multiple domains and applications.

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 grade), 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

---

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

```
 1<Policy PolicyId="demo" RuleCombinationAlgId="first-applicable">
 2 <Target>
 3   <Subjects> <AnySubjects/> </Subjects>
 4   <Resources>
 5    <Resource>
 6     <ResourceMatch MatchId="equal">
 7      <AttributeValue>demo:5</AttributeValue>
 8      <ResourceAttributeDesignator AttributeId="objectid"/>
 9     </ResourceMatch>
10    </Resource>
11   </Resources>
12   <Actions> <AnyAction/></Actions>
13 </Target>
14 <Rule RuleId="1" Effect="Deny">
15  <Target> <Subjects><AnySubject/></Subjects>
16   <Resources> <AnyResource/> </Resources>
17   <Actions>
18    <Action>
19     <ActionMatch MatchId="equal">
20      <AttributeValue>Dissemination</AttributeValue>
21      <ActionAttributeDesignator AttributeId="actionid"/>
22     </ActionMatch>
23    </Action>
24   </Actions>
25  </Target>
26  <Condition FunctionId="not">
27   <Apply FunctionId="at-least-one-member-of">
28    <SubjectAttributeDesignator AttributeId="loginid"/>
29    <Apply FunctionId="string-bag">
30     <AttributeValue>testuser1</AttributeValue>
31     <AttributeValue>testuser2</AttributeValue>
32     <AttributeValue>fedoraAdmin</AttributeValue>
33    </Apply>
34   </Apply>
35  </Condition>
36 </Rule>
37 <Rule RuleId="2" Effect="Permit"/>
38</policy>
```

Fig. 1. An example XACML policy

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 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 earlier. Lines 2-13 define the target of the policy, which indicates that this policy applies only to those access requests of an object "demo:5". The target of Rule 1 (Lines 15-25) further narrows the scope of applicable requests to those asking to perform a "Dissemination" action on

---

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

object "demo:5". Its condition (Lines 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.

3   Access Control Policies and Policy Coverage

Besides XACML, a generic policy language, 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 that 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, and actions as well as other properties of a system, why do we still need $Cond_s$, $Cond_o$, and $Cond_a$ in a rule? The reason is that, although 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 have been scrutinized by the public.

We next start our discussion on policy testing based on the preceding model. The basic idea of policy testing is 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 expected. Any unexpected decision indicates potential errors in the specification of the policy.

If no requests are evaluated against a rule during testing, then potential errors in that rule cannot 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. Recall that if a request satisfies $Cond_s$, $Cond_o$, and $Cond_a$ of a rule, then we say the rule is *applicable* to the request.

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 requests, 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 preceding definition.

Our definition of condition coverage corresponds to clause coverage or condition coverage [33] in program testing. Note that there exist more complicated coverage criteria for logical expressions. For example, in program testing, predicate coverage (also called decision coverage or branch coverage) [33] requires to cover both true and false

of compound conditions in a logical expression. In policy testing, predicate coverage requires that the whole compound condition for a rule needs to be evaluated to be true and false, respectively. In program testing, combinatorial coverage (also called multiple condition coverage) [33] requires to cover each possible combination of outcomes of each condition in a logical expression. In policy testing, combinatorial coverage requires to cover each possible combination of outcomes of each condition for a rule. In our existing approach, we use basic, simple criteria for conditions in rules; in future work, we plan to investigate these more complicated alternatives in terms of their effects on fault-detection capability.

4   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 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 be covered 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 by instrumenting Sun's open source XACML implementation [2]. Sun's implementation facilitates the construction of a PDP. We instrument several methods throughout their implementation that collect policy, rule, and condition information when a policy is loaded into the PDP. Then coverage information is collected and stored in a singleton as requests are evaluated by a PDP against the policy under test.

After the 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 the text file 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 these requests 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

Because manually generating requests for testing policies is tedious, we have developed a technique for randomly generating requests given only the policy under test. The random request generator analyzes the policy under test and generates requests on demand by randomly selecting requests from the set of all combinations of attribute id-value pairs found in the policy. A particular request is represented as a vector of bits. The length of this vector is equal to the number of different attribute values found in the policy set targets, 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.

More specifically, all possible combinations can be represented by integers from 0 to 2^n where n is the number of attribute values found in the policy. 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 modified the random test generation algorithm to ensure that each bit was set to 1 and 0 at least once. In particular, we explicitly set the ith 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 where $i = n + 1, n + 2, ...2n$. This improved algorithm 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 [20]:

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 [27].

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 and 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 Measuring Fault-Detection Capability

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 [15] as a mechanism to compare request sets in terms of fault detection.

Mutation testing [15] has historically been applied to general-purpose programming languages. The program under test is iteratively mutated to produce numerous mutants, each containing one fault. A test input is independently executed on the original program and each mutant program. If the output of a test executed on a mutant differs from the output of the same test executed on the original program, then the fault is detected and the mutant is said to be killed. The fundamental premise of mutation testing as stated by Geist et al. [17] is that, in practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test that also detects that fault. In other words, the ability to detect small, minor faults such as mutants implies the ability to detect complex faults. Because fault detection is the central focus of any testing process, mutation testing provides an external measure of the effectiveness of that process. The higher the percentage of killed mutants, the more effective the test set is at fault detection.

Table 1. Index of mutation operators.

ID	Description
PSTT	Policy Set Target True. The policy set is applied to all requests.
PSTF	Policy Set Target False. The policy set is not applied to any requests.
PTT	Policy Target True. The policy is applied to all requests.
PTF	Policy Target False. The policy is not applied to any requests.
RTT	Rule Target True. The rule is applied to all requests.
RTF	Rule Target False. The rule is not applied to any requests.
RCT	Rule Condition True. The condition always evaluates to true.
RCF	Rule Condition False. The condition always evaluated to false.
CPC	Change Policy Combining Algorithm. Each policy combining algorithm is tried in turn.
CRC	Change Rule Combining Algorithm. Each rule combining algorithm is tried in turn.
CRE	Change Rule Effect. The rule effect is inverted (e.g. permit for deny).

In policy mutation testing, the program under test, test inputs, and test outputs correspond to the policy, requests, and responses, respectively. We first define a set of mutation operators shown in Table 1. Given a policy and a set of mutation operators, a mutator generates a number of mutant policies. Given a request set, we evaluate each request in the request set on both the original policy and a mutant policy. The request evaluation produces two responses for the request based on the original policy and the mutant policy, respectively. If these two responses are different, then we determine that the mutant policy is killed by the request; otherwise, the mutant policy is not killed.

Unfortunately, there are various expenses and barriers associated with mutation testing. The first and foremost is the generation and execution of a large number of mutants. For general-purpose programming languages, the number of mutants is proportional to the product of the number of data references and the number of data objects in the program [34]. For XACML policies, the number of mutants is proportional to the number of policy elements, namely policy sets, policies, targets, rules, conditions, and their associated attributes.

8   Experiment on Request Reduction and Its Effect on Fault Detection

The objective of the experiment is to examine whether the reduced request set is as effective at fault detection as the original request set. Similar to the goals of Hennessy et al. [21] 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 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.*

8.1 Metrics

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. The following metrics are measured for each policy under test, each request set, and each mutation operator.

- *Policy hit percentage*. The policy hit percentage or policy coverage is the number of applicable policies when evaluating the request set divided by the total number of policies.
- *Rule hit percentage*. The rule hit percentage or rule coverage is the number of applicable rules when evaluating the request set divided by the total number of rules.
- *Condition hit percentage*. The condition hit percentage is the number of hit true and hit false conditions when evaluating the request set divided by two times of the total number of conditions.
- *Test count*. The test count is the size of the request set or the number of generated tests. For testing access control policies, a test is synonymous with request.
- *Reduced-test count*. Given a policy and the generated set of requests, the reduced test count is the size of the reduced request set based on policy coverage.
- *Mutant-killing ratio*. Given a request set, the policy under test, and the set of generated mutants, the mutant-killing ratio is the number of mutants killed by the request set divided by the total number of mutants.

Intuitively a set of requests that achieve higher policy coverage are more likely to reveal faults. This notion is easy to understand because a fault in a policy element that is never covered by a request would never contribute to a response and thus a fault in that element cannot possibly be revealed. There is a direct correlation between the test count and the test evaluation time. Furthermore, a low test count is highly desirable because the request-response pairs may need to be inspected manually to verify that the policy specification exhibits the intended policy behavior. An ideal request set should have a low test count, high structural coverage, and high fault-detection capability.

8.2 Results

We used 10 XACML policies collected from three different sources as subjects in our experiment. Table 2 summarizes the basic statistics of each policy. The first column shows the subject names. Columns 2-5 show the numbers of policy sets, policies, rules, and conditions, respectively. Five of the policies, namely `simple-policy`, `codeA`, `codeB`, `codeC`, and `codeD` are examples used by Fisler et al. [16, 18]. The remaining policies are examples of real XACML policies used by Fedora[3]. Fedora is an open source software that gives organizations a flexible service-oriented architecture for managing and delivering digital content. Fedora uses XACML to provide fine-grained access control to the digital content that it manages. The Fedora repository of default and example XACML policies provides a useful resource of realistic subjects.

---

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

Table 2. Policies used in the experiment.

Subject	# PolSet	# Pol	# Rule	# Cond
codeA	5	2	2	0
codeB	7	3	3	0
codeC	8	4	4	0
codeD	11	5	5	0
default-2	1	13	13	12
demo-11	0	1	3	4
demo-26	0	1	2	2
demo-5	0	1	3	4
mod-fedora	1	13	13	12
simple-policy	1	2	2	0

Table 3. Structural coverage, number of requests, and size reduction for each policy.

Subject	Policy Hit	Rule Hit	Cond Hit	#Req	#Reduced Req	Size Reduction
codeA	100.00%	100.00%	-	50	2	0.96
codeB	100.00%	100.00%	-	50	3	0.94
codeC	100.00%	100.00%	-	50	6	0.88
codeD	100.00%	100.00%	-	50	6	0.88
default-2	100.00%	92.31%	75.00%	50	6	0.88
demo-11	100.00%	100.00%	75.00%	50	2	0.96
demo-26	100.00%	100.00%	50.00%	50	1	0.98
demo-5	100.00%	100.00%	75.00%	50	3	0.94
mod-fedora	100.00%	84.62%	58.33%	50	7	0.86
simple-policy	100.00%	100.00%	-	50	4	0.92

We preprocessed each policy to ensure unique policy element identifiers in order to correctly measure structural coverage. Once each policy has been preprocessed, we randomly generate requests for each policy as outlined in Section 5 (we configure that 50 requests are randomly generated for each policy). As these requests are generated and evaluated, we 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 reduction in request-set size, $SizeReduction$, as follows:

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

Columns 2-7 of Table 3 show the three structural coverage metrics, size of the generated request set, the size of the reduced request set, and the computed size reduction for each policy, respectively. A dash indicates that there are no policy elements of that type and thus coverage cannot be computed. The random request set achieves 100% policy coverage for all subjects because it is the most coarse measure of structural coverage. Rule coverage and condition coverage are a finer measure of structural coverage and thus more difficult to achieve with randomly generated requests. The results show
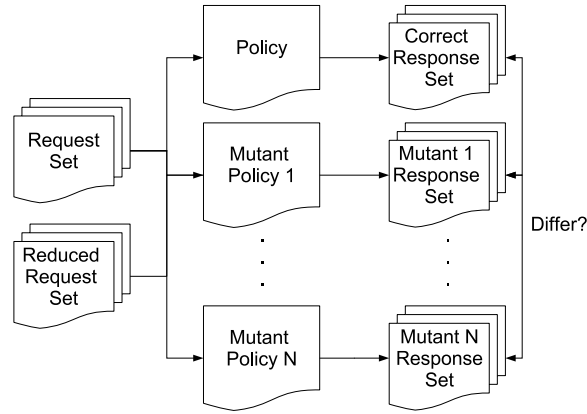
Fig. 2. Overview of fault detection experiment.

that we can achieve an average 92% size reduction for the ten policies. The results suggest that we can indeed greatly reduce the request set size of relatively 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. 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 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:

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

Figure 3 illustrates the average mutant-killing ratios for each request set grouped by subjects. We observe that the mutant-killing ratios across all subjects for the random and reduced random request sets are quite similar. Unfortunately the mutant-killing ratio is still low when considering the high structural coverage. The observation indicates that a stronger criteria is needed. Specifically the average mutant-killing ratios for the Random, and Reduced Random request sets are 51.8% and 42.1%, respectively. Table 4 lists the mutant-kill ratios in tabular format along with the computed capability reduction. In summary, we observe a 92% reduction in size of the requests while only a 23% reduction in fault detection capability.
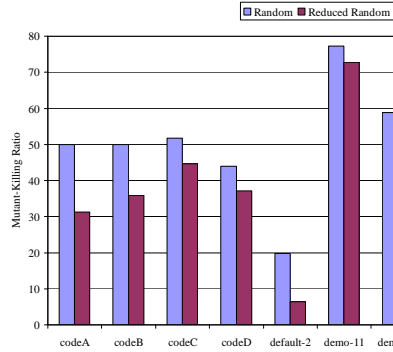
Fig. 3. Mutant-killing ratios for all operators by subjects.

Table 4. Mutant-killing ratios and capability reduction for each request set and each policy.

Subject	Random	Reduced Random	Capability Reduction
codeA	50.00%	31.25%	37.50%
codeB	50.00%	35.87%	28.26%
codeC	51.79%	44.64%	13.79%
codeD	43.92%	37.16%	15.38%
default-2	19.75%	6.37%	67.74%
demo-11	77.27%	72.73%	5.88%
demo-26	58.82%	52.94%	10.00%
demo-5	78.26%	73.91%	5.56%
mod-fedora	25.48%	22.29%	12.50%
simple-policy	62.50%	43.75%	30.00%

In summary, the results indicate that structural coverage is indeed correlated to fault-detection capability. But structural coverage is still not strong enough to achieve an acceptable level of fault detection. Note that the structural coverage investigated in this experiment is essentially equivalent to statement coverage in general-purpose programming languages. In future work, we plan to investigate stronger criteria that correspond to path coverage. We expect these stronger criteria to be much more effective at achieving higher killing ratios.

8.3   Threats to Validity

The threats to external validity primarily include the degree to which the subject policies, mutation operators, coverage metrics, and test sets are representative of true practice. These threats could be reduced by further experimentation on a wider type and larger number of policies and an larger number of mutation operators. In particular, lower level mutation operators are needed to operate on the subject, resource, and action attributes found in various policy elements. Currently the proposed mutation operators operate only on higher level policy elements. The threats to internal validity are

Table 5. 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

instrumentation effects that can bias our results such as faults in Sun's XACML implementation as well as faults in our own policy mutator, policy coverage measurement tool, and request generator.

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 [6] and a conference paper review system's policy and its requests developed by Zhang et al. [41].

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. [41] 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 these conditions, 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 measurement of rule hit percentage.

We fed 374 requests in the XACML conformance test suite to the coverage-measurement tool. Table 5 shows the reported statistics of policy coverage. 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.

---

[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.

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%. 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 5, 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 that either a true or false condition would be hit. We inspected their requests and found that a subject's age was specified twice and their conditions access the subject's age. When evaluating the conditions, PDP encountered an error and returned a decision of `Indeterminate`; therefore, neither true or false condition was 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 the coverage-measurement tool 11 requests for the conference paper review system's policy [41], 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 the 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   Related Work

Much work has been done in the modeling and verification of access control policies. A variety of policy languages and models have been proposed. Some of them are generic [1,14,25,26,37] while others are designed for specific applications [7,11,36,38] or data models [8,9,19,29].

One important aspect of policy verification is to formally check general properties of access control policies, such as inconsistency and incompleteness [10,25,29,31]. In the

former case, an access request can be both accepted and denied according to the policy, while in the latter case the request is neither accepted nor denied. Although efficient algorithms have been proposed to perform such verification for specific systems [24, 26], this problem can be intractable or even undecidable when dealing with policies that involve complex constraints.

Besides the verification of general properties, several tools have been developed to verify properties for XACML policies [1]. Hughes and Bultan translated XACML policies to the Alloy language [23] and checked their properties using the Alloy Analyzer. Fisler et al. [16] developed a tool called Margrave that uses multi-terminal binary decision diagrams [13] to verify user-specified properties and perform change-impact analysis. Zhang et al [42] developed a model-checking algorithm and tool support to evaluate access control policies written in *RW* languages, which can be converted to XACML [41]. 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 [43]. 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 [4, 5] or conditions [12] 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 [40]. 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.

Within 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 [21] 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 [39] defined coverage of SQL queries and developed a tool to automate the measurement. Kapfhammer and Soffa [28] 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.

11   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 testing 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 [6] and a conference reviewing system [41]. 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. XACML.NET. http://mvpos.sourceforge.net/, 2005.
4. 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.
5. 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.
6. A. Anderson. XACML 1.1 committee specification conformance tests. http://www.oasis-open.org/committees/xacml/ConformanceTests/, 2002.
7. R. J. Anderson. A security policy model for clinical information systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 30–43, 1996.
8. E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo. A logical framework for reasoning on data access control policies. In *Proc. 12th IEEE Computer Security Foundations Workshop*, pages 175–189, 1999.
9. E. Bertino, S. Castano, and E. Ferrari. On specifying security policies for web documents with an XML-based language. In *Proc. 6th ACM Symposium on Access Control Models and Technologies*, pages 57–65, Chantilly, VA, May 2001.
10. P. Bonatti, S. Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proc. ACM Conference on Computer and Communication Security*, pages 164–173, Athens, Greece, November 2000.

11. C. Bussler and S. Jablonski. Policy resolution for workflow management systems. In *Proc. Hawaii International Conference on System Science*, pages 831–840, Maui, Hawaii, January 1995.
12. J. Chang and D. J. Richardson. Structural specification-based testing: automated support and experimental evaluation. In *Proc. 7th ESEC/FSE*, pages 285–302, 1999.
13. 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*, pages 1–15, 1993.
14. 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.
15. 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.
16. 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.
17. R. Geist, A. J. Offutt, and F. Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5):55–558, 1992.
18. M. M. Greenberg, C. Marks, L. A. Meyerovich, and M. C. Tschantz. The soundness and completeness of Margrave with respect to a subset of XACML. Technical Report CS-05-05, Department of Computer Science, Brown University, 2005.
19. P. Griffiths and B. Wade. An authorization mechanism for a relational database systems. *ACM Transactions on Database Systems*, 1(3), 1976.
20. 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.
21. 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*, pages 104–113, November 2005.
22. 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.
23. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. 8th ESEC/FSE*, pages 62–73, 2001.
24. T. Jaeger, X. Zhang, and F. Cacheda. Policy management using access control spaces. *ACM Transactions on Information and System Security*, 6(3), 2003.
25. S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proc. 1997 IEEE Symposium on Security and Privacy*, pages 31–42, 1997.
26. S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 474–485, 1997.
27. D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974.
28. G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th ESEC/FSE*, pages 98–107, 2003.
29. M. Kudo and S. Hada. XML document security based on provisional authorization. In *Proc. ACM Conference on Computer and Communication Security*, pages 87–96, Athens, Greece, November 2000.
30. M. Lorch, D. Kafura, and S. Shah. An XACML-based policy management and authorization service for globus resources. In *Proc. International Workshop on Grid Computing*, pages 208–212, Phoenix, AZ, Nov 2003.
31. E. C. Lupu and M. Sloman. Conflict in policy-based distributed systems management. *IEEE Transaction on Software Engineering*, 25(6):852–869, 1999.

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

33. G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

34. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, October 2000.

35. 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.

36. T. Ryutov and C. Neuman. Representation and evaluation of security policies for distributed system services. In *Proc. DARPA Information Survivability Conference and Exposition*, pages 172–183, January 2000.

37. R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based aministration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, Feb. 1999.

38. E. Sirer and K. Wang. An access control language for web services. In *Proc. 7th ACM Symposium on Access Control Models and Technologies*, pages 23–30, Monterey, CA, June 2002.

39. 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.

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

41. 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.

42. 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.

43. 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.