Toward Systematic Testing of Access Control Policies

Evan Martin
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
eemartin@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

ABSTRACT

To facilitate managing access control in a system, access control policies are increasingly written in specification languages such as XACML. A dedicated software component called a Policy Decision Point (PDP) interprets the specified policies, receives access requests, and returns responses to inform whether access should be permitted or denied. To increase confidence in the correctness of specified policies, policy developers can conduct policy testing by supplying typical test inputs (requests) to the PDP and subsequently checking test outputs (responses) against expected ones. Unfortunately, manual testing is tedious and few tools exist for automated testing of XACML policies.

In this paper, we present our work toward a framework for systematic testing of access control policies. The framework includes components for policy coverage definition and measurement, request generation, request evaluation, request set minimization, policy property inference, and mutation testing. This framework allows us to evaluate various criteria for test generation and selection, investigate mutation operators, and determine a relationship between structural coverage and fault-detection capability. We have implemented the framework and applied it to various XACML policies. Our experimental results offer valuable insights into choosing mutation operators in mutation testing and choosing coverage criteria in test generation and selection.

1. INTRODUCTION

Access control is one of the most fundamental and widely used security mechanisms. It controls which principals such as users or processes have access to which resources in a system. To facilitate managing and maintaining access control, access control policies are increasingly written in specification languages such as XACML [1] and Ponder [5]. Whenever a principal requests access to a resource, that request is passed to a software component called a *Policy Decision Point* (PDP). A PDP evaluates the request against the specified access control policies, and permits or denies the request accordingly.

Assuring the correctness of policy specifications is becoming an important and yet challenging task, especially as access control policies become more complex and are used to manage a large amount of sensitive information organized into sophisticated structures. Identifying discrepancies between policy specifications and their intended function is crucial because correct implementation and enforcement of policies by applications is based on the premise that the policy specifications are correct. As a result, policy specifications must undergo rigorous verification and validation through systematic testing to ensure the policy specifications truly encapsulate the desires of the policy authors.

Proceedings of MASPLAS'06
Mid-Atlantic Student Workshop on Programming Languages and Systems
Rutgers University, April 29, 2006

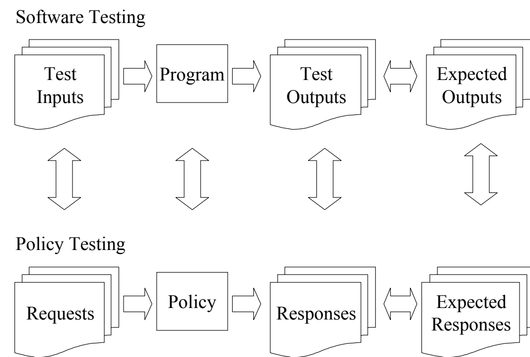

Figure 1: Mapping of traditional software testing to policy testing.

Software testing aims at efficiently detecting and correcting errors in software through dynamic execution. Errors in policy specifications may also be discovered by leveraging existing techniques for software testing and applying them to policy testing. The similarities of traditional software testing and policy testing are illustrated in Figure 1. In policy testing, test inputs are access requests and test outputs are access responses. The execution of test inputs occurs as requests are evaluated by the PDP against the access control policies under test. Policy authors can inspect request-response pairs to check whether they are expected. Access control policies are often tested with manually defined access requests so that policy authors may check the PDP's responses against expected ones [3]. Unfortunately, current policy testing practice tends to be a manual, ad hoc process. With such a process, it is questionable that high confidence can be gained on the correctness of access control policies.

Mutation testing [6] has historically been applied to general-purpose programming languages in measuring the quality of a test suite. In this paper, we present our prevous work toward a framework for systematic policy testing. In particular we have developed a framework for automated mutation testing of access control policies [17]. In the framework, we define a set of new mutation operators for XACML policies. We also develop a tool that automatically seeds a policy under test with faults by applying these mutation operators, thereby producing numerous mutant policies. We leverage a change-impact analysis tool to detect equivalent mutants among generated mutants. We determine whether a mutant policy is killed by a request by comparing the responses for the request based on the original policy and mutant policy. Our framework can be applied on XACML policies together with our previous tools of test generation, test selection, and structural coverage measurement for access control policies [16, 19]. We perform an experiment that uses mutation testing to evaluate structural coverage criteria for test generation and test selection in terms of fault-detection capabilities. Our experimental results offer valuable insights into choosing mutation operators in mutation testing and choosing coverage criteria in test generation and selection. Finally we present our preliminary results on inferring properties of policies via machine learning applied to request-response pairs [18].

The rest of the paper is organized as follows. We first present related work in Section 2 and a brief introduction to XACML in Section 3. We then present policy coverage, test generation, and test minimization in Sections 4, 5, and 6. We then present our work on mutation testing of access control policies in Section 7. Section 8 describes the experiment where we apply the automated test generation, test minimization, coverage measurement, and mutation testing on various XACML policies. Finally we present our preliminary results on policy property inference via machine learning in Section 9 and conclude with Section 10.

2. RELATED WORK

To help ensure the correctness of policy specifications, researchers and practitioners have developed formal verification tools for policies [8,13,23]. Fisler et al. [8] developed a tool called Margrave that can verify XACML [1] policies against properties, if they are specified, and perform change-impact analysis on two versions of policies when properties are not specified. Margrave performs property verification by automatically generating concrete counter-examples in the form of specific requests that illustrate violations of the specified properties. Similarly, change-impact analysis is performed by automatically generating specific requests that reveal semantic differences between two versions of a policy. Although verification tools such as Margrave are valuable, it is sometimes beyond the capabilities of these tools to verify complex access control policies because of the tools' limited support for various XACML features. Furthermore, user-specified properties are often not available [8]. Our mutation testing framework leverages Margrave's strengths for generating requests and detecting equivalent mutants.

Although various coverage criteria [24] for software programs exist, only recently have coverage criteria for access control policies been proposed in our previous work [19]. Policy coverage criteria are needed to measure how well policies are tested and which parts of the policies are not covered by the existing tests. In our previous work [19], we have defined policy coverage and developed a policy coverage measurement tool. Because it is tedious for developers to manually generate test inputs for policies, and manually generated tests are often not sufficient for achieving high policy coverage, we have also developed several techniques of test generation. The first one iterates over all possible requests for a given policy, if its domain set is finite. The second one is a random test generation tool that randomly generates tests for XACML policies [19]. The third technique [16] is a novel framework that automatically generates high-quality tests based on a change-impact analysis tool such as Margrave [8]. Because the number of automatically generated tests is often too large for manual inspection, we developed a request-reduction tool that greedily selects a minimal set of tests for achieving the same policy coverage as the original set of tests. Our new automated mutator allows us to quickly evaluate test generators and techniques of test selection in terms of fault-detection capabilities.

3. XACML

The eXtensible Access Control Markup Language (XACML) is an XML based syntax used to express policies, requests, and responses. This general-purpose language for access control policies is an OASIS (Organization for the Advancement of Structured Information Standards) standard [1] that describes both a language for policies and a language for requests or responses of access control decisions. The policy language is used to describe general access control requirements and is designed to be extended to include new functions, data types, combining logic, etc.

The five basic elements of XACML policies are `PolicySet`, `Policy`, `Rule`, `Target`, and `Condition`. A policy set is simply a container that holds other policies or policy sets. A policy is expressed through a set of rules. With multiple policy sets, policies, and rules, XACML must have a way to reconcile conflicting rules. A collection of combining algorithms serves this function [1]. Each algorithm defines a different way to combine multiple decisions into a single decision. Both *policy* combining algorithms and *rule* combining algorithms are provided. Seven standard combining algorithms are provided but user-defined combining algorithms are also allowed [2].

To aid in matching requests with the appropriate policies, XACML provides a target [1], which is basically a set of simplified conditions for the subject, resource, and action that must be met for a policy set, policy, or rule to apply to a given request. Once a policy or policy set is found to apply to a given request, its rules are evaluated to determine the response.

Finally we have attributes, attribute values, and functions. Attributes are named values of known types that describe the subject, resource, and action of a given access request [1]. A request is formed of attributes that will be compared to attributed values in a policy to make the access decisions. Attribute values from a request are resolved through two mechanisms: the `AttributeDesignator` and the `AttributeSelector` [1]. The former lets the policy specify an attribute with a given name and type, whereas the latter allows a policy to look for attribute values through an XPath query.

4. COVERAGE MEASUREMENT

Sun has developed an open source, pure Java implementation of the XACML standard [2]. This implementation supports parsing of policies, requests, and responses; determining applicable policies for a given request; evaluating requests against a set of policies; as well as implementing standard attribute types, functions and policy combining algorithms.

Our previous work [19] defines three policy coverage metrics and presents a tool called Poco for measuring policy coverage. Poco automatically measures three structural coverage metrics: policy hit percentage, rule hit percentage, and condition hit percentage. Poco is essentially an instrumented version of Sun's XACML implementation that automatically collects policy coverage informa-

tion as requests are evaluated against a policy. Poco acts as a PDP that accepts access requests and returns access decisions.

As discussed in Section 3, XACML policies have three major entities: policies, rules for each policy, and conditions for each rule. Policy coverage is quantified for each of these entities and is defined as follows [19]:

- *Policy hit percentage*. A policy is hit by a request if the policy is applicable to the request. If all the conditions in the policy's target are satisfied by the request, then the policy is applicable to 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. If the policy is applicable to the request and all the conditions in the rule's target are satisfied by the request, then the policy's rule is also applicable to 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: the true condition and false condition. 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 the number of total conditions.

5. TEST GENERATION

This section presents two approaches to test generation for access control policies. The first is a simple random request generation technique and the second is a more sophisticated approach that leverages change-impact analysis to achieve high structural coverage.

5.1 Random Test Generation

Because manually generating requests for testing policies is tedious, our previous work [19] developed a random test generation tool for policies. The tool 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 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.

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

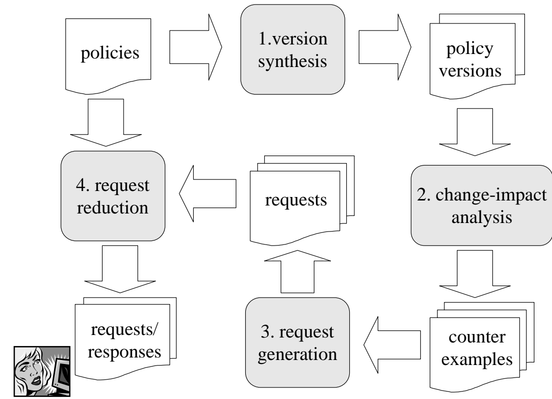

Figure 2: An overview of the framework for test generation based on change-impact analysis.

the $(i - n)^{th}$ bit to 0. 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 $2n$.

5.2 Test Generation via Change-Impact Analysis

To automatically generate high-quality test suites for access control policies, our previous work has developed a novel framework based on change-impact analysis [16]. Our framework receives a set of policies under test and outputs a set of tests in the form of request-response pairs for developers to inspect their correctness. Figure 2 is a high-level illustration of this framework. The framework consists of four major components: version synthesis, change-impact analysis, request generation, and request reduction. The key notion of the framework is to synthesize two versions of the given policies in such a way that test coverage targets (e.g., certain policies, rules, and conditions) are encoded as the differences of the two synthesized versions. Then a change-impact analysis tool can be leveraged to generate counterexamples to witness these differences, thus covering the test coverage targets. The framework generates tests (in the form of requests) based on the generated counterexamples. We implemented this framework in a tool called Cirg that leverages Poco and Margrave to automatically generate test suites with high structural coverage [16].

6. TEST MINIMIZATION

The number of generated requests can be large for complex policies. In such cases it is infeasible for policy authors to inspect each request-response pair; therefore, we need to reduce the number of requests for inspection without incurring substantial loss in fault detection capability.

We have defined request reduction problem [19] 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.

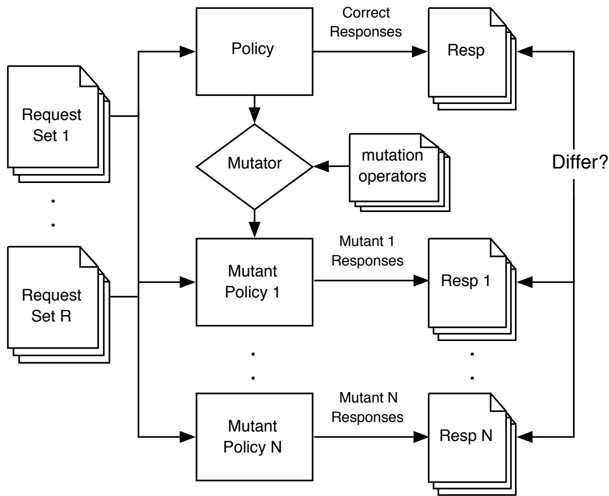

Figure 3: Overview of our framework for policy mutation testing.

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 say 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 [14].

7. MUTATION TESTING

This section presents our framework for policy mutation testing. We first introduce the general concept of mutation testing and describe mutation testing for access control policies. We then present how to detect equivalent mutants among generated mutants. Finally, we present a set of mutation operators used by our automated policy mutator.

Mutation testing [6] 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. [9] 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.

In policy mutation testing, the program under test, test inputs, and test outputs correspond to the policy, requests, and responses, respectively. An overview of our framework for policy mutation testing is illustrated in Figure 3. In the framework, we first define a set of mutation operators, whose details are described in Section 7.2. 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 [20]. 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.

7.1 Equivalent-Mutant Detection

Cost of mutation testing also includes detection of equivalent mutants [20]. Although there are syntactic differences between an equivalent mutant and the program under test, the mutant is semantically equivalent to the original one. In other words, the mutant will produce the same result as the original one for all test inputs and thus provides no benefit. Equivalent-mutant detection provides a mechanism to better evaluate mutation operators and more efficiently perform mutation testing because computational resources will not be wasted in evaluating test inputs or comparing test outputs for equivalent mutants. Detecting such mutants in software is generally intractable [7] and historically has been done by hand [20] but using a change-impact analysis tool such as Margrave [8] allows us to detect equivalent mutants among generated mutants. We originally believed equivalent-mutant detection to be an important efficiency improvement though we found in practice that evaluating requests and comparing responses to be computationally cheaper than performing change-impact analysis with Margrave. Furthermore, limitations of Margrave prevented the detection of equivalent mutants for mutation operators on conditions and some combining algorithms.

7.2 Mutation Operators

Previous studies [12, 15] have been conducted to investigate the types and effectiveness of various mutation operators for general-purpose programming languages; however, these mutation operators often do not directly apply to mutating policies. This section describes the chosen mutation operators for XACML policies. An index of the mutation operators is listed in Table 1 and their details are described as below.

1. *Policy Set Target True (PSTT)*. Ensure that the policy set is applied to all requests by removing the `<Target>` tag of each `PolicySet` element. The number of mutants created by this operator is equal to the number of `PolicySet` elements with a `<Target>` tag. The number of equivalent mutants created depends on the specific policy under test.

2. *Policy Set Target False (PSTF)*. Ensure that the policy set is never applied to a request by modifying the `<Target>` tag of each `PolicySet` element. The number of mutants created by this operator is equal to the number of `PolicySet` elements. The number of equivalent mutants created depends on the specific policy under test.

Table 1: Index of mutation operators.

ID	Description
PSTT	Policy Set Target True
PSTF	Policy Set Target False
PTT	Policy Target True
PTF	Policy Target False
RTT	Rule Target True
RTF	Rule Target False
RCT	Rule Condition True
RCF	Rule Condition False
CPC	Change Policy Combining Algorithm
CRC	Change Rule Combining Algorithm
CRE	Change Rule Effect
RMPS	Remove Policy Set
RMP	Remove Policy
RMR	Remove Rule

3. *Policy Target True (PTT)*. Ensure that the policy is applied to all requests simply by removing the `<Target>` tag of each `Policy` element. The number of mutants created by this operator is equal to the number of `Policy` elements with a `<Target>` tag. The number of equivalent mutants created depends on the specific policy under test.

4. *Policy Target False (PTF)*. Ensure that the policy is never applied to a request by modifying the `<Target>` tag of each `Policy` element. The number of mutants created by this operator is equal to the number of `Policy` elements. The number of equivalent mutants created depends on the specific policy under test.

5. *Rule Target True (RTT)*. Ensure that the rule is applied to all requests simply by removing the `<Target>` tag of each `Rule` element. The number of mutants created by this operator is equal to the number of `Rule` elements with a `<Target>` tag. The number of equivalent mutants created depends on the specific policy under test.

6. *Rule Target False (RTF)*. Ensure that the rule is never applied to a request by modifying the `<Target>` tag of each `Rule` element. The number of mutants created by this operator is equal to the number of `Rule` elements. The number of equivalent mutants created depends on the specific policy under test.

7. *Rule Condition True (RCT)*. Ensure that the condition always evaluates to `True` simply by removing the condition of each `Rule` element. The number of mutants created by this operator is equal to the number of `Rule` elements with a `<Condition>` tag. The number of equivalent mutants created depends on the specific policy under test.

8. *Rule Condition False (RCF)*. Ensure that the condition always evaluates to `False` by manipulating the condition value or the condition function. The number of mutants created by this operator is equal to the number of `Rule` elements. The number of equivalent mutants created depends on the specific policy under test.

9. *Change Policy Combining Algorithm (CPC)*. Try all possible policy combining algorithms. This high-level mutation may change the way that various policies interact. This operator is only meaningful if there is more than one `Policy` element in the policy under test. Currently there are six policy combining algorithms implemented in Sun's XACML implementation [2] but four of these algorithms semantically reduce to two, leaving only four policy combining algorithms, namely deny-overrides, permit-overrides, first-applicable, and only-one-applicable. The number of mutants created by this operator for policies with more than one `Policy` element is three and zero otherwise. The number of equivalent mutants created depends on the specific policy under test.

10. *Change Rule Combining Algorithm (CRC)*. Try all possible rule combining algorithms. This high-level mutation may change the way that various rules interact. This operator is only meaningful if there is more than one `Rule` element in the policy under test. Currently there are five rule combining algorithms implemented in Sun's XACML implementation [2] but four of these algorithms semantically reduce to two, leaving only three rule combining algorithms, namely deny-overrides, permit-overrides, and first-applicable. The number of mutants created by this operator for policies with more than one `Rule` element is two and zero otherwise. The number of equivalent mutants created depends on the specific policy under test.

11. *Change Rule Effect (CRE)*. Invert each rule's `Effect` by changing `Permit` to `Deny` or `Deny` to `Permit`. The number of mutants created by this operator is equal to the number of rules in the policy. This operator should never create equivalent mutants unless a rule is unreachable, in which case the rule should probably be removed.

12. *Remove Policy Set (RMPS)*. If there is more than one `PolicySet` element, then we remove each policy set in turn. The number of created mutants is equal to the number of `PolicySet` elements in the entire policy. This operator only creates equivalent mutants if the removed `PolicySet` is unreachable or redundant in which case the policy set should probably be removed.

13. *Remove Policy (RMP)*. If there is more than one `Policy` element, then we remove each policy in turn. The number of created mutants is equal to the number of `Policy` elements in the entire policy. This operator creates equivalent mutants only if the removed `Policy` is unreachable or redundant, in which case the policy should probably be removed.

14. *Remove Rule (RMR)*. If there is more than one `Rule` element, then we remove each rule in turn. The number of created mutants is equal to the number of `Rule` elements in the entire policy. This operator creates equivalent mutants only if the removed `Rule` is unreachable or redundant in which case it should probably be removed. (Note that we do not have a *Remove Condition (RMC)* mutation operator, because this mutation operator has the exactly same semantic as RCT.)

8. MUTATION EXPERIMENT

This section presents the experiment that we conducted to evaluate our policy mutator and the defined mutation operators. The policy mutator uses the defined mutation operators to automatically seed the policy under test with faults for generating mutant policies. These mutant policies are then used to evaluate request sets to determine the mutant-killing ratios. This process provides a measure of quality for each request set in terms of fault-detection capability. Because two of these request sets are generated based on the

structural coverage of the policy, we can find correlations between structural coverage and fault-detection capability. We first describe the experiment's objective and measures as well as the experiment instrumentation. We then present and discuss the experimental results and finally describe threats to validity.

8.1 Objective and Measures

The objective of the experiment is to investigate the following questions:

1. How strong is the correlation between structural coverage and fault-detection capability? More specifically, does test selection based on structural coverage criteria produce request sets with high fault-detection capability?
2. What are the individual characteristics of each mutation operator? Are some more difficult to kill than others? Are some easily killed by request sets selected based on structural coverage criteria?

To help answer these questions, we collect several metrics to compare the request generation techniques based on change-impact analysis, random request generation, and the minimized random request set based on structural coverage. 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 policies involved in 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 rules involved in evaluating the request set divided by the total number of rules.
- *Condition hit percentage*. The condition hit percentage is the number of conditions involved in 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 tests generated by the chosen test-generation technique. 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 because a large request set would take longer to evaluate than a smaller set. 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 Instrumentation

In the experiment, we used the policy mutator for generating mutants, the Cirg tool for test generation [16] based on change-impact analysis, a random request generation tool [19], a policy coverage measurement tool [19] for test selection, and Margrave [8] for limited equivalent-mutant detection.

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
conference	0	1	15	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

We collected policies from several sources as subjects in our experiment. Each policy is preprocessed to ensure unique policy element identifiers in order to correctly measure structural coverage. Once each policy has been preprocessed, we can apply a request generation technique to generate tests. We compare three requests sets. The first one is generated by Cirg based on change-impact analysis. The second one is randomly generated. The third one is actually a subset of the second greedily selected to ensure equivalent structural coverage.

The random test generation technique requires only the complete policy. The technique parses the policy and enumerates all possible attribute id-value pairs. This set is represented as a vector of bits and each bit is randomly set to 0 or 1, which indicates the absence or presence of the corresponding attribute id-value pair in the generated request as described in Section 5.1. We generate exactly 50 random requests for each subject. Finally, we greedily select requests from this set based on structural coverage. Doing so allows us to directly measure the reduction in fault detection capability when selecting requests based on structural coverage.

The test generation technique based on change-impact analysis uses only one of the variants of version synthesis described in our previous work [16]. The policy versions are essentially equivalent to the mutants generated with the CRE operator. We use Margrave's API to perform a change-impact analysis on the original policy and each of the policy versions. Based on the counterexamples produced by Margrave, the request generator generates requests. Exactly one request is generated from each version.

We used 11 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 3-5 show the numbers of policy sets, policies, rules, and conditions, respectively. The `conference`[1] policy is a slightly modified version of the policy used by Zhang et al. [22]. The `<Condition>` tags were removed so Sun's XACML implementation could evaluate the requests. This policy relies on custom functions implemented in the PDP that interact with a database at runtime for request evaluation. Sun's XACML implementation supports only the standard functions and so it failed to evaluate requests properly. Once the relevant conditions were removed from the policy, requests were evaluated successfully. Although these modifications changed the semantics of the policy, it is structurally similar and thus suitable for the experiment. Five of the policies, namely `simple-policy`, `codeA`, `codeB`, `codeC`, and `codeD` are

[1] `http://www.cs.bham.ac.uk/~mdr/research/projects/05-AccessControl/`

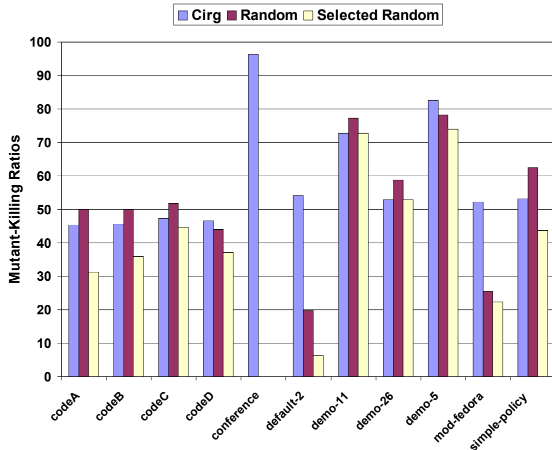

Figure 4: Mutant-killing ratios for all operators by subjects.

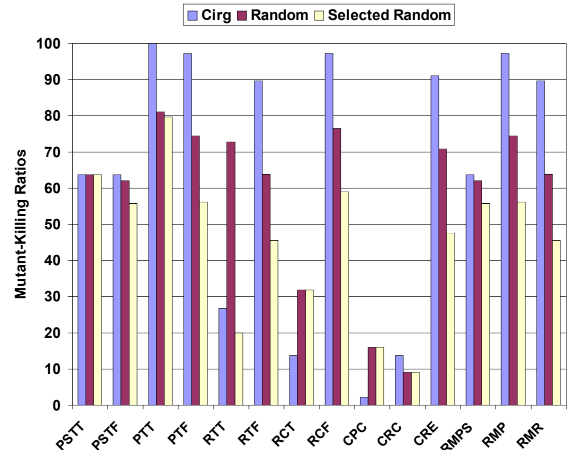

Figure 5: Mutant-killing ratios for all subjects by operators.

examples used by Fisler et al. [8, 10]. The remaining policies are examples of real XACML policies used by Fedora[2]. 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 it manages. The Fedora repository of default and example XACML policies provided a useful resource of realistic subjects.

8.3 Results

Table 3 summarizes the structural coverage metrics for each policy and each request set. We do not show the minimized random request set because it has equivalent coverage as its superset. Each row of the table corresponds to a particular policy and each column group corresponds to a request set. Within each column group, we show the policy, rule, and condition coverage percentages. N/A indicates that there are no policy elements of that type and thus coverage cannot be computed. Both test generation techniques achieve 100% policy coverage for almost all subjects because it is the most coarse measure of structural coverage. Cirg achieves only 50% condition coverage because the generation technique does not attempt to evaluate the condition as true *and* false but merely covers the condition's rule once. However, for policy and rule elements, Cirg is at least as good as random generation at achieving high structural coverage.

Figure 4 illustrates the average mutant-killing ratios for each request set grouped by subjects. By comparing these results with those in Table 3, we observe that there is indeed a correlation between structural coverage and fault detection capability. One example is the conference policy; the structural coverage for the two random request sets is zero and, as expected, the mutant-killing ratio is also zero. Similarly we observe that the mutant-killing ratios across all subjects for the random and selected 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 Cirg, Random, and Selected Random request sets are 59%, 47%, and 38%, respectively.

Figure 5 illustrates the average mutant-killing ratios for each request set grouped by mutation operators. Recall that the CPC and CRC mutation operators exploit the way that various policies and various rules interact, respectively. These mutation operators have less than 11% mutant-killing ratios. The observation indicates that these operators produce mutants that are particularly difficult to kill. Conversely, RMPS, PSTF, and PSTT have over 60% killing ratios, and RMR, RTF, CRE, PTF, RMP, RCF, and PTT have at least 90% killing ratios. By inspecting the results, we found that some of these mutation operators are redundant. For example, RMPS and PSTF have identical killing ratios across all request sets. This observation indicates that these two operators are semantically equivalent. Similar results are observed for the pairs of RMP-PTF and RMR-RTF. In retrospect, these mutation operators should be semantically equivalent because a target that is always evaluated to false is semantically equivalent to removing the element that the target applies to.

We provided the original policy and each mutant policy to Margrave's change-impact analysis feature to perform equivalent-mutant detection. If Margrave finds counter-examples that illustrate differences between the policies, then they *must not* be equivalent. Unfortunately Margrave supports only a subset of XACML features; therefore, the converse does not hold, resulting in potential false positives. In other words, if Margrave does *not* find counter-examples for a particular mutant, then the mutant *may* or *may not* be equivalent. In our experiment, Margrave identified less than 1% of all mutants as potentially equivalent. Furthermore, these potentially equivalent mutants occurred only for the CPC and CRC mutation operators. Performing equivalent mutation detection is costly, taking approximately 45 minutes for the whole experiment. When considering the low percentage of detection, potential for false positives, and high computational cost, we feel other means of equivalent mutant detection are needed.

In summary, the results indicate that although structural coverage is indeed correlated to fault-detection capability, structural coverage is 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. Similar to the findings in mutation testing of general-purpose programming languages, we found that equivalent-mutant detection is expensive and some mutation operators are redundant, indicating a subset of mutation operators may be sufficient for mutation testing.

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

Table 3: Structural coverage achieved by each request set.

	Random Request Set					Cirg			
Subject	Pol %	Rule %	Cond %	# Req	# Min Req	Pol %	Rule %	Cond %	# Req
codeA	100	100	N/A	50	2	100	100	N/A	2
codeB	100	100	N/A	50	3	100	100	N/A	3
codeC	100	100	N/A	50	6	100	100	N/A	4
codeD	100	100	N/A	50	6	100	100	N/A	5
conference	0	0	N/A	50	0	100	100	N/A	15
default-2	100	92.31	75	50	6	100	100	50	13
demo-11	100	100	75	50	2	100	100	50	2
demo-26	100	100	50	50	1	100	100	50	2
demo-5	100	100	75	50	3	100	100	50	3
mod-fedora	100	84.62	58.33	50	7	84.62	84.62	33.33	11
simple-policy	100	100	N/A	50	4	100	100	N/A	2

8.4 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 instrumentation effects that can bias our results such as faults in Sun's XACML implementation, faults in Margrave's API and/or its limitations, as well as faults in our own policy mutator, policy coverage measurement tool, and request generators.

9. PROPERTY INFERENCE

Again our primary objective is to efficiently identify discrepancies between the policy specification and the true desires of the policy authors. In other words, we wish to reveal faults in the policy specification. We help a user identify these discrepancies or bugs by finding specific requests that are likely bug-exposing. We first observe the policy's behavior by probing it with several automatically generated requests. These observations are used as input, in the form of request-response pairs, to a particular class of machine learning algorithms called classification learning. The output of the machine learning algorithms is essentially a summary of the policy in the form of inferred properties that may *not* be true for all requests but are true for *most* requests. We do not wish to recreate the policy in its entirety through these inferred properties but merely capture the general policy behavior in order to help identify special cases. The rationale is that the policy specification is mostly correct and that the bug-exposing requests represent a small percentage. Under this rationale, any request that violates the inferred properties are special cases or requests that result in responses that deviate from the policy's normal behavior. These special cases are identified as being likely bug-exposing and warrant manual inspection. We have integrated Sun's XACML implementation [2] and a collection of machine learning algorithms for data mining tasks [21] into a tool that implements our approach through request generation, request evaluation, and policy property inference.

We infer policy properties by applying machine learning on request-response pairs. Our current tool leverages Weka [21], a collection of machine learning algorithms for data mining tasks. Weka contains tools for pre-processing, classification, regression, clustering, association rules, and visualization. In general, data mining is defined as the process of discovering patterns in data such as explicit knowledge structures (i.e., structural descriptions) [21].

In our research context, we are mostly interested in the knowledge structures acquired as a mechanism to infer general and not necessarily universally true properties of the policy. Machine learning techniques are frequently used to gain insight into the structure of their data rather than to make predictions for new cases [21]. We use knowledge structures generated from a genre of machine learning algorithms called *classification learning* to summarize the results of request-response pairs thereby expressing the policy in a different and often more concise way.

As requests are evaluated against the policy, our tool appends relevant information about the request-response pairs to a data file in a particular format being used as training data for Weka [21]. Weka mines the request-response pairs to find and describe structural patterns. These structural patterns are described in the form of rules or properties that are simple conditional expressions or properties that classify requests into four response types: `Permit`, `Deny`, `NotApplicable`, and `Indeterminate`. These rules are useful for manual inspection and for identifying corner cases. Because the rules produced by the classification learning algorithms are statistically true, it is likely of interest to the user to inspect the requests that violate those rules. If violating requests exist in the training data, then they are identified by Weka as misclassified instances. If no violating request has been generated by the request factory, then it is possible to translate the rule into a property and use an existing property verification tool [8, 13, 23] to generate a request or set of requests that violate the inferred property.

9.1 Preliminary Results

We applied our tool on an access control policy of a central grades repository system for a university, which was earlier used by Fisler et al. [8] to illustrate policy verification. Because the policy defines small, finite sets of subjects, resources, and actions, we use the `AllComboReqFactory` to generate the entire set of possible requests. With 3 subjects, 2 resources, and 2 actions, the request factory generated $2^7 = 128$ different requests. Unfortunately 56 of the combinations produced invalid requests that resulted in `Indeterminate` responses, 54 evaluated to `NotApplicable` responses, 10 evaluated to `Deny` responses, and 8 evaluated to `Permit` responses.

We used the Prism classification algorithm [4] to generate rule sets using two different sets of training data. The first set used the output from all 128 requests. In the second set we removed all instances with the response of `Indeterminate` or `NotApplicable`. Although the performance of the partial data set appears similar to

```
1.1 If Faculty = 1
    and (Receive = 1 or Assign = 1)
    and (ExternalGrades = 1 or InternalGrades = 1)
    then Permit
1.2 If Student = 1
    and Receive = 1
    and ExternalGrades = 1 then Permit
1.3 If Student = 1
    and Assign = 1
    and ExternalGrades = 1 then Deny
1.4 If True then Deny
```

Figure 6: Rules in the actual XACML policy.

```
2.1 If Faculty = 1 then Permit
2.2 If Student = 1
    and InternalGrades = 0
    and Receive = 1 then Permit
2.3 If TA = 1 then Deny
2.4 If Student = 1
    and InternalGrades = 1 then Deny
2.5 If Student = 1
    and Receive = 0 then Deny
```

Figure 7: Rules generated by the Prism classification algorithm on the partial data set.

that of the full data set, the number of rules is smaller and more relevant for the partial data set. The full data set produces 30 rules whereas the partial data set produces the 5 rules shown in Figure 7.

For comparison purposes, we have translated the rules in the actual XACML policy to the form shown in Figure 6. By comparing Figure 6 and Figure 7, we see that the inferred properties do indeed summarize the policy. However, because there are misclassified instances, we know that the inferred properties are *not* universally true. Note that Rules 6.1 and 6.2 are equivalent to Rules 7.1 and 7.2, respectively. Rules 7.4 and 7.5 are intuitively correct because a student should not have access to internal grades or have assign permissions for any resource.

An error in the policy specification was discovered after we investigated the misclassified requests. Recall that those misclassified requests represent instances in which the policy produces responses that are inconsistent with the responses of similar requests. The rationale is that these special cases are likely bug-exposing requests. The request in question is one in which a `Student` wishes to `Receive` and `Assign` the resource of `ExternalGrades`. The classification model *correctly* classifies the response as `Deny` but the policy evaluates the request to `Permit`. The policy authors did not intend for a student to have permissions to assign their own grade as shown by Rule 6.3. This error is the same discrepancy found by Fisler et al. [8], which is a result of a subtlety of the XACML language. The root cause of the problem is that XACML allows an arbitrary number of values for a given attribute. This example illustrates that the investigation of misclassified requests can lead to the discovery of errors in policy specifications.

This simple example has shown that even with a small number of request-response pairs, machine learning can be a valuable tool for discovering and summarizing the basic properties of a policy. We suspect that its value and power will increase as the complexity and size of the policy grows because the inferred properties can summarize and aggregate the complex rules specified in the expressed policy. Further experimentation with a broader range of classification algorithms on large, complex policies is still required in future work to further assess the approach.

10. CONCLUSION

We have developed several tools that contribute to a framework toward systematic policy testing. We have defined policy coverage, implemented a tool to measure it, and used it as a criteria for test selection. We have developed a random test generation technique as well as more complicated techniques based on change-impact analysis. We have evaluated the test generation and selection techniques in terms of structural coverage and fault detection capability. We have developed an automated mutation testing framework for access control policies. In this framework, we have defined a set of mutation operators. We have implemented a mutator that generates a number of mutant policies based on the defined mutation operators. We evaluate each request in a given 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. We have also leveraged a change-impact analysis tool to detect equivalent mutants among generated mutants. We have conducted an experiment on various XACML policies to evaluate the mutation operators as well as request generation and selection techniques in terms of fault-detection capabilities. Our experimental results show that although structural coverage is a strong indicator of fault-detection effectiveness, it is far from optimal. The shortcomings of test selection based on structural coverage are highlighted by mutation operators that exploit how different policy elements interact. Moreover, careful test generation and selection techniques can substantially reduce the size of the test suite while incurring a relatively low loss of fault-detection capability. Finally we present an approach to policy property inference via machine learning and some preliminary results of the application of the technique on an access control policy. The results indicate that machine learning can be a valuable tool for discovering and summarizing the basic properties of a policy although further experimentation with a broader range of classification algorithms on larger, more complex policies is still required to further assess the approach.

11. ACKNOWLEDGMENTS

We would like to thank Ting Yu for discussions that help improve the work described in this paper.

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

[4] J. Cendrowska. Prism: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27(4):349–370, 1987.

[5] 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.

[6] 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.

[7] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[8] 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.

[9] 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.

[10] 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.

[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*, pages 104–113, 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. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974.

[15] 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.

[16] E. Martin and T. Xie. Automated test generation for access control policies via change-impact analysis. In *Under Submission*. `http://www.csc.ncsu.edu/faculty/xie/publications/cirg.pdf`.

[17] E. Martin and T. Xie. Automated mutation testing of access control policies. In *Submitted to the* 22nd *IEEE Interational Conference on Software Maintenance (ICSM 2006)*, September 2006. `http://www.csc.ncsu.edu/faculty/xie/publications/policymutation.pdf`.

[18] E. Martin and T. Xie. Inferring access-control policy properties via machine learning. In *Proc. of International Workshop on Policies for Distributed Systems and Networks (POLICY 2006)*, June 2006.

[19] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Submitted to the IEEE/ACM Interational Conference on Automated Software Engineering (ASE 2006)*, 2006. `http://www.csc.ncsu.edu/faculty/xie/publications/xacmlcov.pdf`.

[20] 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.

[21] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[22] 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.

[23] 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.

[24] 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.