A Fault Model and Mutation Testing of Access Control Policies

Evan Martin and Tao Xie
Dept. of Computer Science, North Carolina State University
Raleigh, NC, USA
eemartin@ncsu.edu, xie@csc.ncsu.edu

ABSTRACT

To increase confidence in the correctness of specified policies, policy developers can conduct policy testing by supplying typical test inputs (requests) and subsequently checking test outputs (responses) against expected ones. Unfortunately, manual testing is tedious and few tools exist for automated testing of access control policies.

We present a fault model for access control policies and a framework to explore it. The framework includes mutation operators used to implement the fault model, mutant generation, equivalent-mutant detection, and mutant-killing determination. This framework allows us to investigate our fault model, evaluate coverage criteria for test generation and selection, and determine a relationship between structural coverage and fault-detection effectiveness. We have implemented the framework and applied it to various policies written in XACML. Our experimental results offer valuable insights into choosing mutation operators in mutation testing and choosing coverage criteria in test generation and selection.

Categories and Subject Descriptors: D.2.5 [Testing and Debugging]: Testing tools

General Terms: Reliability.

Keywords: Fault Model, Mutation Testing, Test Generation, Access Control Policies.

1. INTRODUCTION

Access control is one of the most fundamental and widely used security mechanisms, especially in web applications. 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 [6]. 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 that the policy specifications truly encapsulate the desires of the policy authors.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

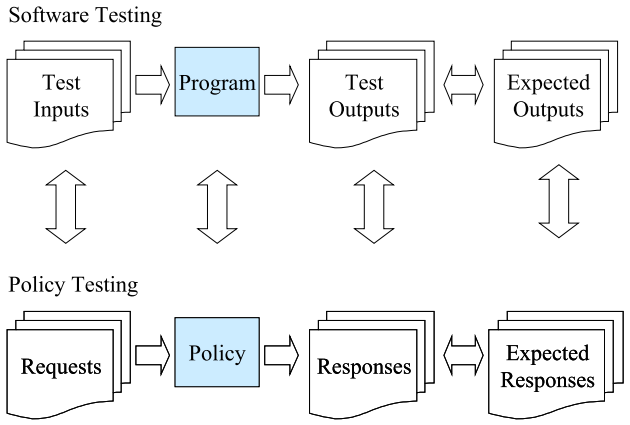

Figure 1: Analogy between traditional software testing and policy testing.

Software testing is an important and practical technique to efficiently detect errors in complex software systems. Errors in policy specifications may also be discovered by leveraging existing techniques for software testing. Figure 1 illustrates the analogy between software testing and policy testing. In policy testing, test inputs are access requests and test outputs are access responses. The execution of a test input occurs as a request is evaluated by the PDP against the access control policy under test. Policy authors can inspect request-response pairs to check whether they are as expected. Like software verification and testing techniques, formal policy verification and testing techniques are complementary means to achieve the same goal.

Mutation testing [7] iteratively modifies the program under test to produce numerous faulty versions (called mutants), each containing one fault. Mutation testing has historically been applied to general-purpose programming languages in measuring the quality of a test suite. In this paper, we propose a fault model for access control policies. We present a new framework that implements this fault model to facilitate automated mutation testing of access control policies. In the framework, we define a set of new mutation operators for XACML policies based on the fault model; these mutation operators describe modification rules for modifying a policy to generate faulty policies (called mutant policies). We also develop a new 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; these equivalent

mutants are syntactically different from the original policy but are semantically equivalent to it. 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 previous tools that we have developed for test generation, test selection, and structural coverage measurement for access control policies [23–25]. 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.

The rest of the paper is organized as follows. We first present related work in Section 2 followed by an illustrative example in Section 3. Section 4 provides some background information. We then present our fault model for access control policies in Section 5 followed by the corresponding mutation testing framework in Section 6. Section 7 describes the experiment where we apply the framework on various XACML policies. Finally we conclude with Section 8.

2. RELATED WORK

To help ensure the correctness of policy specifications, researchers and practitioners have developed formal verification tools for policies [9, 18, 39]. Fisler et al. [9] developed a tool called Margrave that can verify XACML [1] policies against properties, if properties 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 counterexamples 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 must be expressed formally (not in natural language) and as a result are often not available [9]. Our mutation testing framework leverages Margrave's change-impact analysis feature to detect a subset of equivalent mutants.

Although various coverage criteria [40] for software programs exist, only recently have coverage criteria for access control policies been proposed [25]. 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 [25], 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 test generation techniques. 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 [25] that randomly generates tests for XACML policies. The third technique [24] is a novel framework that automatically generates high-quality tests based on a change-impact analysis tool such as Margrave [9]. 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 fault model and corresponding automated mutator allow us to quickly evaluate test generators and techniques of test selection in terms of fault-detection capability. Techniques [8] have been proposed

```
<Policy Id="demo" RuleCombAlgId="first-applicable">
 <Target>
   <Subjects> <AnySubjects/> </Subjects>
   <Resources>
    <Resource>
     <ResourceMatch MatchId="equal">
      <AttrValue>demo:5</AttrValue>
      <ResourceAttrDesignator AttrId="objectid"/>
     </ResourceMatch>
    </Resource>
   </Resources>
   <Actions> <AnyAction/></Actions>
 </Target>
 <Rule RuleId="1" Effect="Deny">
 <Target> <Subjects><AnySubject/></Subjects>
  <Resources> <AnyResource/> </Resources>
  <Actions>
   <Action>
    <ActionMatch MatchId="equal">
     <AttrValue>Dissemination</AttrValue>
     <ActionAttrDesignator AttrId="actionid"/>
    </ActionMatch>
   </Action>
  </Actions>
 </Target>
 <Condition FunctionId="not">
  <Apply FunctionId="at-least-one-member-of">
   <SubjectAttrDesignator AttrId="loginid"/>
   <Apply FunctionId="string-bag">
    <AttrValue>testuser1</AttrValue>
    <AttrValue>testuser2</AttrValue>
    <AttrValue>fedoraAdmin</AttrValue>
   </Apply>
  </Apply>
 </Condition>
</Rule>
<Rule RuleId="2" Effect="Permit"/>
</Policy>
```

Figure 2: An example XACML policy

to leverage mutation testing to automatically generate and/or reduce test sets for general purpose programming languages. We take similar approaches for test generation and reduction techniques for testing access control policies.

3. EXAMPLE

We first present the syntax of XACML and give an example of a mutation operator and how that operator is used to generate mutant policies to facilitate mutation testing.

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 and 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 [1] serves this function. 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.

XACML provides attributes, attribute values, and functions. Attributes are named values of known types that describe the subject, resource, and action of a given access request. A request is formed of attributes that will be compared to attribute 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. Functions can work on any combination of attribute values and can return any kind of attribute value supported in the system; these functions are used to compare values to make access decisions. For example, one may use a role-based access control policy that provides various levels of access based on the subject's `role` attribute. A `SubjectAttributeDesignator` in the `Condition` of a `Rule` can be used to compare the `role` attribute against expected values to determine the level of access allowed for the given `Subject`.

Figure 2 shows an example XACML policy, which is revised and simplified from a sample Fedora policy (to be used in our experiment described in Section 7). This policy has one policy element, which in turn contains two rules. The rule combining algorithm is "first-applicable", meaning that the decision of the first applicable rule encountered during evaluation is returned. Lines $2-13$ define the policy's target, which indicates that this policy only applies to those access requests of a resource "demo:5". The target of Rule 1 (Lines $15-25$) further narrows the scope of applicable requests to those requesting to perform a "Dissemination" action on resource "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 combining algorithm of the policy (Line 1), a request applicable to the policy should be permitted.

In mutation testing, mutation operators are used to generate mutant policies. A mutant policy is identical to the policy under test but with one syntactic difference automatically inserted based on the used mutation operator. For example, one mutation operator is Change-Rule Effect (CRE). Applying CRE to the example policy in Figure 2 generates two mutant policies. One mutant policy has the rule decision on Line 14 as permit and the other mutant policy has the rule decision on Line 37 as deny. After the mutant policies are generated, each test (in this case an access request) in a test set is executed both on the policy under test and each mutant policy. If a test output (in this case an access response) differs between the policy under test and a mutant policy, then the mutant is said to be killed (i.e., the fault is exposed). The more mutants a test set kills, the more effective that test set is in terms of fault-detection capability. We use mutation testing as a quality measure for various test generation and selection techniques.

4. BACKGROUND

This section presents background information for our framework, including a description of our previous work [23–25] on structural coverage measurement, request generation, and request selection.

4.1 Coverage Measurement

We have defined three types of policy structural coverage for each of the three major entities in an XACML policy: policies, rules, and conditions.

- *Policy coverage*. A policy is covered by a request if the policy is applicable to the request *and* the policy contributes to the decision; in other words, all the conditions in the policy's target are satisfied by the request and the PDP has yet to fully resolve the decision for the given request. Policy coverage is the number of covered policies divided by the number of total policies.
- *Rule coverage.* A rule for a policy is covered by a request if the rule is also applicable to the request *and* the rule contributes to the decision; in other words, the rule is applicable to the request and all the conditions in the rule's target are satisfied by the request and the PDP has yet to fully resolve the decision for the given request. Rule coverage is the number of covered rules divided by the number of total rules.
- *Condition coverage.* 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 covered by a request if the rule is covered by the request and the condition is evaluated to be true. A false condition for a rule is covered by a request if the rule is covered by the request and the condition is evaluated to be false. Condition coverage is the number of covered true conditions and covered 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 use a measurement tool [25] implemented by instrumenting Sun's open source XACML implementation [2]. Sun's implementation facilitates the construction of a PDP. Several methods throughout their implementation collect policy, rule, and condition information when a policy is loaded into the PDP. Then coverage information is collected and stored as requests are evaluated by the PDP against the policy under test.

4.2 Random Test Generation

Because manually generating requests for testing policies is tedious, we have developed a random test generation tool for policies [25]. 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. 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 i^{th} bit to 1 for the first n generated requests where $i = 1, 2, ..., n$. Similarly, for the next n requests, we explicitly set the $(i-n)^{th}$ bit to 0 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 $2n$.

4.3 Test Generation via Change-Impact Analysis

To automatically generate high-quality test suites for access control policies, we have developed a framework based on change-impact analysis [24]. The 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 for correctness. 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 [24] that leverages Margrave [9] to automatically generate test suites with high structural coverage.

4.4 Test Selection

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 selection or reduction problem [25] similar to the test minimization problem for program testing [14]:

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 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 [19]. We employ a greedy algorithm that removes a request from a request set if and only if the request does not increase any of the coverage metrics that are achieved by previously evaluated requests in the request set.

5. FAULT MODEL

This section presents a fault model for access control policies and a set of mutation operators that implement that model. In general, a fault model is an engineering model of something that could go wrong in the construction or operation of a piece of equipment, structure, or software. In our case, we are modeling things that could go wrong when constructing an access control policy. We use this fault model to measure the fault-detection effectiveness of automatic test generation and selection techniques. Any fault results in a semantic change in the policy but we broadly categorize faults as being semantic or syntactic as follows: syntactic faults are a result of simple typos whereas semantic faults are associated with the logical constructs of the policy language.

Syntactic faults are easier to make and consist of simple typos that result in a semantically faulty policy. Syntactic faults may result in syntactically incorrect policies but we assume that basic static analysis tools exist to check for such inconsistencies. For example, in XACML, an XML schema definition (XSD) can be used to check for obvious flaws in XACML syntax. In addition, syntactic faults that do not violate the XSD can occur due to typos in attribute values. If the set of attribute values for a given attribute-id is finite and can be enumerated, then a domain-specific XSD can be written to check for correct attribute values. For example, consider Line 8 of the example XACML policy in Figure 2. The attribute value "demo:5" can accidentally contain a typo (e.g., "demo5") causing the target to be incorrectly specified. The fault may allow unauthorized access to the "demo:5" resource. This fault can be detected by a domain-specific XSD that is written to ensure that the attribute value is a valid resource. Not all syntactic faults can be detected by an XSD. For example, if "demo:5" is accidentally written as "demo:6", which is a valid resource, then the XSD cannot detect this fault. Typos in attribute values can cause the conditions found in the target and condition elements to evaluate to true or false when they should not. We define and implement several mutation operators that emulate these faults.

Semantic faults are more elusive than syntactic faults because semantic faults involve incorrect use of the logical constructs of the policy specification language. For XACML policies, these logical constructs include policy combining algorithms, rule combining algorithms, policy evaluation order, rule evaluation order, and various functions found in the condition. Unlike some syntactic faults, semantic faults can not be detected by an XSD. For an example of a semantic fault, consider the rule combining algorithm "first-applicable" on Line 1 of the example XACML policy in Figure 2. A logical mistake would be to use the "permit-overrides" rule combining algorithm. The result of this mistake is severe because all access requests would be permitted. The final rule on Line 37 applies to all requests and has a permit decision; thus, using the "permit-overrides" rule combining algorithm ensures all access requests will be permitted.

5.1 Mutation Operators

Mutation operators describe modification rules for modifying access control policies to introduce faults into the policies. Previous studies [15,21] 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 that implements our fault model. An index of the mutation operators is listed in Table 1 and their details are described below. The first eight mutation operators emulate syntactic faults because these mutation operators manipulate the predicates found in the target and condition

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

elements. In particular, PSTT, PSTF, PTT, PTF, RTT, RTF, RCT, and RCF emulate syntactic faults as simple typos in the policy set, policy, and rule target elements as well as the condition elements which result in the predicates found in those elements to always evaluate to true or false. The last three mutation operators, CPC, CRC, and CRE, emulate semantic faults because they manipulate the logical constructs of XACML policies.

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.

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.

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.

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.

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.

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.

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.

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.

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 semantically different standard 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.

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.

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, a strong indication of an error in the policy specification.

6. MUTATION TESTING FRAMEWORK

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.

6.1 Mutation Testing

Mutation testing [7] 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. [11] 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 5.1. 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

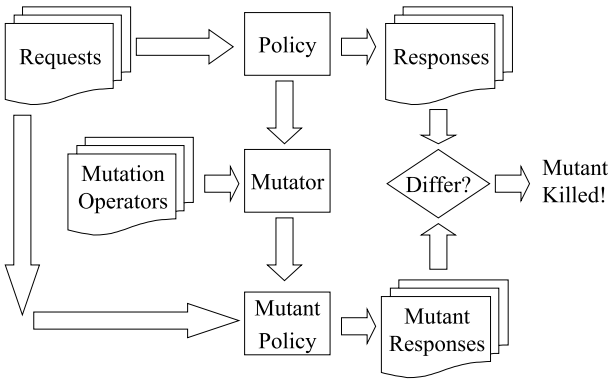

Figure 3: Overview of our framework for policy mutation testing.

objects in the program [33]. 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.

Techniques to reduce the cost of mutation testing fall into two basic approaches: test with fewer mutants and test smarter. The test fewer approach simply involves generating and/or executing fewer mutants; selective mutation and mutant sampling both fall into this category. *Constrained mutation* [33, 36] later refined into *selective mutation* [28, 29, 33] is an approximation technique that tries to select only mutants that are truly distinct from other mutants. Results show that 5 out of 22 mutation operators are *key* operators and these 5 provide almost the same coverage with cost reductions of four times with small programs and up to 50 times for larger programs [28, 29]. *Mutant sampling*, first proposed by Acree [3] and Budd [4], involves randomly selecting a subset of mutant programs, which are then evaluated. Results from Wong [35] show that a 10% random sample of mutants is only 16% less effective than a full set in ascertaining fault-detection effectiveness. Another sampling approach [34] selects mutant programs based on a Bayesian sequential probability ratio test until sufficient evidence has been collected to determine that a statistically appropriate sample size has been reached.

Various test smarter approaches involve optimizations for specific computer architectures [5, 20, 26, 31] and techniques that exploit the classic space-time trade-off [10]. For example, *weak mutation* [17] is an approximation technique that reduces execution costs by comparing the internal states of the mutant and original programs instead of their output at program termination. Weak mutation has been discussed theoretically [16, 27, 37], studied empirically [12, 22, 32], and probed with variants that differ on exactly when the program states should be compared [27, 37]. Weak mutation has been shown to generate tests that were almost as effective as tests generated with strong mutation and that at least 50% or more of the execution time was saved [30, 32].

6.2 Equivalent-Mutant Detection

Cost of mutation testing also includes detection of equivalent mutants [33]. 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 [8] and historically has been done by hand [33] but using a change-impact analysis tool such as Margrave [9] 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. EXPERIMENT

This section presents the experiment that we conducted to evaluate our fault model. The policy mutator implements the fault model by using 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.

7.1 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 reduced request sets with low loss of 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 selected 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 selected or 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.

7.2 Instrumentation

In the experiment, we used the policy mutator for generating mutants, the Cirg tool [24] for test generation based on change-impact analysis, a random request generation tool [25], a policy coverage measurement tool [25] for test selection, and Margrave [9] for limited equivalent-mutant detection.

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 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 4.2. 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 in Cirg [24]. 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 2-5 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. [38]. 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 examples used by Fisler et al. [9, 13]. 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 that it manages. The Fedora repository of default and example XACML policies provided a useful resource of realistic subjects.

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

7.3 Results

Table 3 summarizes the number of requests and structural coverage metrics for each policy and each request set. We do not show the coverage metrics for 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 as well as the number of requests. N/A indicates that there are no policy elements of that type and thus coverage cannot be computed. The columns "Rand", "Sel Rand", and "Cirg" indicate the number of requests in the Random, Selected Random, and Cirg request sets, respectively. 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. We unexpectedly notice that Cirg exhibits worse policy and rule coverage than the random technique on the `mod-fedora` policy. Upon further examination of that policy we find that two of the policies within the policy set are identical. Furthermore, each of these policies contains exactly one rule. As a result, there was no change-impact when that rule was removed and so Cirg did not generate a request to cover that rule or its containing policy.

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

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

Table 3: Structural coverage achieved by each request set.

	Random Request Set					Cirg			
Subject	Pol %	Rule %	Cond %	# Rand	# Sel Rand	Pol %	Rule %	Cond %	# Cirg
codeA	100.00%	100.00%	N/A	50	2	100.00%	100.00%	N/A	2
codeB	100.00%	100.00%	N/A	50	3	100.00%	100.00%	N/A	3
codeC	100.00%	100.00%	N/A	50	6	100.00%	100.00%	N/A	4
codeD	100.00%	100.00%	N/A	50	6	100.00%	100.00%	N/A	5
conference	0.00%	0.00%	N/A	50	0	100.00%	100.00%	N/A	15
default-2	100.00%	92.31%	75.00%	50	6	100.00%	100.00%	50.00%	13
demo-11	100.00%	100.00%	75.00%	50	2	100.00%	100.00%	50.00%	2
demo-26	100.00%	100.00%	50.00%	50	1	100.00%	100.00%	50.00%	2
demo-5	100.00%	100.00%	75.00%	50	3	100.00%	100.00%	50.00%	3
mod-fedora	100.00%	84.62%	58.33%	50	7	84.62%	84.62%	33.33%	11
simple-policy	100.00%	100.00%	N/A	50	4	100.00%	100.00%	N/A	2
average	90.91%	88.81%	30.30%	50.00	3.64	98.60%	98.60%	21.21%	5.64

Table 4: Mutant-kill results achieved by each request set.

		Random		Selected Random		Cirg	
Subject	# Mut	# Kill	Kill %	# Kill	Kill %	# Kill	Kill %
codeA	64	32	50.00%	20	31.25%	29	45.31%
codeB	92	46	50.00%	33	35.87%	42	45.65%
codeC	112	58	51.79%	50	44.64%	53	47.32%
codeD	148	65	43.92%	55	37.16%	69	46.62%
conference	82	0	0.00%	0	0.00%	79	96.34%
default-2	157	31	19.75%	10	6.37%	85	54.14%
demo-11	22	17	77.27%	16	72.73%	16	72.73%
demo-26	17	10	58.82%	9	52.94%	9	52.94%
demo-5	23	18	78.26%	17	73.91%	19	82.61%
mod-fedora	157	40	25.48%	35	22.29%	82	52.23%
simple-policy	32	20	62.50%	14	43.75%	17	53.13%
average			47.07%		38.27%		59.00%

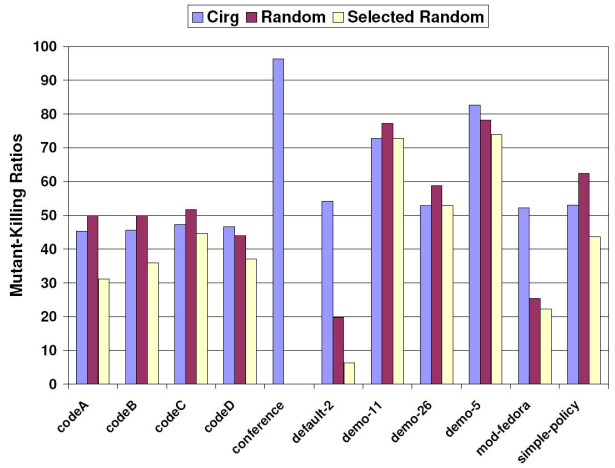

Figure 4: Mutant-killing ratios for all operators by subjects.

Table 4 summarizes the number of mutants generated, the number of mutants killed, and the computed mutant-kill ratios for each policy and each request set. Each row corresponds to a policy. The second column "# Mut" denotes the number of mutants generated for each policy. Each column group corresponds to a request set and lists the number of mutants killed and the computed mutant-killing ratios. The same data is illustrated in Figure 4. 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. Similar to statement coverage in software, low coverage indicates a deficiency in the test set but high coverage does not necessarily indicate a high-quality test set in terms of fault-detection capability. This observation indicates that stronger criteria are needed. The average mutant-kill ratios are given in the last row of Table 4 for the Random, Selected Random, and Cirg request sets as 47.07%, 38.27%, and 59%, respectively. This indicates that while Random request generation may be able to achieve adequate coverage for some policies, Cirg outperforms the Random technique in terms of fault-detection capability.

Figure 5 illustrates the average mutant-killing ratios for each request set grouped by mutation operators. Recall that the CPC and CRC mutation operators are faults that 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. Another explanation is that these mutation operators produce a large number of equivalent mutants causing an arti-

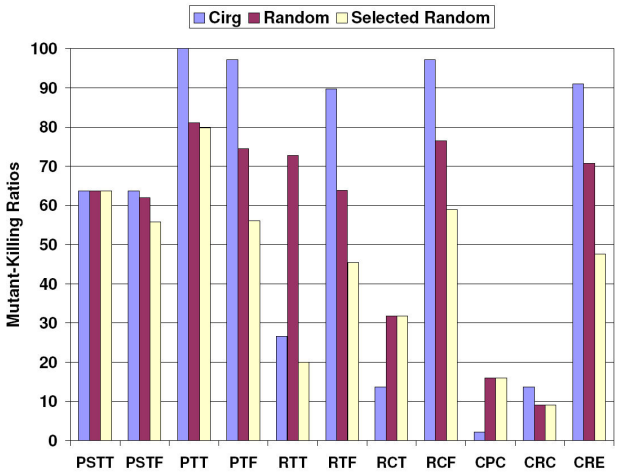

Figure 5: Mutant-killing ratios for all subjects by operators.

ficial lowering of the mutant-killing ratio. Indeed this explanation may be the case because policy and rule combining algorithms are only relevant when there is a high degree of interaction between rules within a policy or policies within a policy set. Conversely, PSTT and PSTF have over 60% killing ratios, and PTT, PTF, RTF, RCF, and CRE have at least 90% killing ratios. The similar mutant-killing ratios across these mutation operators indicate that there is no significant difference between the fault types in terms of difficulty of detection.

We provided the original policy and each mutant policy to Margrave's change-impact analysis feature to perform equivalent-mutant detection. If Margrave finds counterexamples 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 counterexamples 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 mutant-killing ratios. Similar to the findings in mutation testing of general-purpose programming languages, we found that equivalent-mutant detection is expensive.

7.4 Threats to Validity

The threats to external validity primarily include the degree to which the policies, fault model, 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 a 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. Additional mutation operators are needed to exploit policy and rule evaluation order as well as the numerous functions that may be specified in the condition. 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.

8. CONCLUSION

We have proposed a fault model for access control policies and developed an automated mutation testing framework that implements that model. 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.

9. 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. T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, 1980.

[4] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, 1980.

[5] B. Choi and A. P. Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20:135–152, 1993.

[6] 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.

[7] 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.

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

[9] 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.

[10] V. N. Fleyshgakker and s. N. Weiss. Efficient mutation analysis: A new approach. In *Proc. International Symposium on Software Testing and Analysis*, pages 185–195, 1994.

[11] 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.

[12] M. R. Girgis and M. R. Woddward. An integrated system for program testing using weak mutation and data flow analysis. In *Proc. 8th International Conference on Software Engineering*, pages 313–319, 1985.

[13] 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.

[14] 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.

[15] 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.

[16] J. R. Horgan and A. P. Mathur. Weak mutation is probably strong mutation. Technical Report SERC-TR-83-P, Software Engineering Research Center - Purdue University, December 1990.

[17] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[18] 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.

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

[20] E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on SIMD machines. *IEEE Trans. Softw. Eng.*, 17(5):403–423, 1991.

[21] 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.

[22] B. Marick. The weak mutation hypothesis. In *Proc. 4th Symposium on Software Testing, Analysis, and Verification*, pages 190–199, 1991.

[23] E. Martin and T. Xie. Inferring access-control policy properties via machine learning. In *Proc. International Workshop on Policies for Distributed Systems and Networks*, pages 235–238, June 2006.

[24] E. Martin and T. Xie. Automated test generation for access control policies via change-impact analysis. In *Proc. 3rd International Workshop on Software Engineering for Secure Systems*, May 2007.

[25] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. 8th International Conference on Information and Communications Security*, pages 139–158, December 2006.

[26] A. P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. Technical Report SERC-TR-14-P, Software Engineering Research Center - Purdue University, 1988.

[27] L. J. Morell. A theory of fault-based testing. *IEEE Trans. Softw. Eng.*, 16(8):844–857, 1990.

[28] A. J. Offut, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proc, 15th Internation Conference on Software Engineering*, pages 100–107, May 1993.

[29] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5:99, April 1996.

[30] A. J. Offutt and S. D. Lee. How strong is weak mutation? In *Proc. 4th Symposium on Software Testing, Analysis, and Verification*, pages 200–213, 1991.

[31] A. J. Offutt, R. Pargas, S. V. Fichter, and P. Khambekar. Mutation testing of software using a MIMD computer. In *Proc. International Conference on Parallel Processing*, pages 257–266, 1992.

[32] A. J. Offutt and s. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20:337–344, 1994.

[33] 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.

[34] M. Sahinoglu and E. H. Spafford. A bayes sequential statistical procedure for approving software products. In *Proc. IFIP Conference on Approving Software Products*, pages 43–56, 1990.

[35] W. E. Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, 1993.

[36] W. E. Wong, M. E. Delamaro, J. Maldonado, and A. P. Mathur. Constrained mutation in c programs. In *Proc. 8th Brazilian Symposium on Software Engineering*, pages 439–452, October 1994.

[37] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*, pages 152–158, 1988.

[38] 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.

[39] 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.

[40] 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.