Assessing Quality of Policy Properties in Verification of Access Control Policies*

Evan Martin Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
{eemartin, txie}@ncsu.edu

Vincent C. Hu
National Institute of Standards and Technology
Gaithersburg, MD, USA
vincent.hu@nist.gov

ABSTRACT

As sensitive information is increasingly available online through various distributed protocols, the need for carefully controlling access to that information is increasingly important. Control means not only preventing the leakage of data but also permitting access to necessary information.

To facilitate managing, maintaining, and analyzing access control, access control policies are often specified in domain-specific, declarative languages. To increase confidence in the correctness of specified policies, policy authors can use policy verification tools to formally verify policies against a set of properties, which are often manually specified. Policy verification is an important technique for high assurance of the correct specification of access control policies. The effectiveness of the verification is directly related to the quality of the properties, i.e., how comprehensively the properties cover various behaviors of the policy and thus assure correctness of these behaviors once verified.

In this paper, we propose a novel approach called Mutaver to assess the quality of properties specified for a policy and, in doing so, the quality of the verification itself. Similar to the way mutation testing is used to assess the quality of a test suite in terms of fault-detection capability, we propose mutation verification to assess the quality of a set of properties. Given a policy and a set of properties, we first mutate the policy to generate various mutant policies, each with a single fault. We then verify whether the properties hold for each mutant policy. If the properties fail to hold for a given mutant policy, then the verification process accurately identifies the fault in the mutant policy. We have implemented a mutation verification tool for `XACML` and applied it to policies and properties from a real-world software system.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.2.8 [**Software Engineering**]: Metrics—*process metrics*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*

General Terms

Reliability, Security, Verification

Keywords

Mutation testing, access control policies, verification, `XACML`

*This work is supported in part by NSF grant CNS-0716579.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2008 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

1. INTRODUCTION

Access control is one of the fundamental and widely used security mechanisms for software and hardware resources, especially for distributed systems. It controls which principals such as users or processes have access to which resources in a system. It is problematic for programmers to hard code access control policies in the program itself because (1) tracing the policy becomes difficult as the program is maintained and evolves, (2) it is difficult to share the policy across systems, and (3) automated reasoning about the policy becomes difficult since its logic is entangled with other parts of the program. Furthermore, it is unnecessarily difficult to analyze and reason about a policy implemented in programs written in a general-purpose programming language, which undoubtedly contains many operations, data, and language-specific nuances unrelated to the policy itself. As a result, a growing trend has emerged towards writing separate access-control-policy specifications in standardized, declarative languages such as `XACML` [1] and Ponder [7]. These policy specifications are integrated with various components of a system in a standardized manner. At runtime, a software component called a Policy Decision Point (PDP) evaluates an access request against the specified access control policies, and permits or denies the request accordingly.

Implementing and maintaining these policies are important and yet challenging tasks, especially as access control policies become more complex and are used to manage a large amount of distributed and sensitive information. Identifying discrepancies between policy specifications and their intended function is crucial because correct implementation and enforcement of policies by applications are based on the premise that the policy specifications are correct. As a result, policy specifications must undergo rigorous verification and validation to ensure that the policy specifications truly encapsulate the desires of the policy developers.

Testing is an important and practical technique to detect errors in complex software systems. In policy testing [24–28], test inputs are access requests and test outputs are access responses. Policy authors can inspect request-response pairs to check whether they are as expected. Testing, while useful, suffers from the hindrances of requiring oracles to alleviate manual effort in test-output inspection and may not necessarily be exhaustive. Exhaustive testing, while

possible, is impractical due to the sheer number of test inputs whose outputs still require manual inspection. The value of both availability and privacy of information demands a high degree of confidence in a policy specification. Policy developers would thus benefit from complementing testing with more exhaustive formal verification methods. Property verification [10, 17, 19, 20, 38, 45] consumes a policy and a property, and determines whether the policy satisfies the property. Unfortunately, property elicitation is rarely complete and identifying missing properties is difficult.

One relatively straightforward idea is to assess which policy elements (such as rules or their conditions) are *accessed* during the verification of the policy against the given property and then consider these accessed policy elements as being *covered* by the property. Then if verifying a property does not access certain policy elements, it seems natural to consider that the property does not *cover* the behavior related to these policy elements. In fact, the preceding idea is related to structural coverage measurement [46] in traditional software testing for measuring the quality of test inputs (i.e., how well the test inputs cover various parts of the software under test).

The underlying mechanism of this preceding idea is also related to the instant consistency checking approach proposed by Egyed [9] for UML models. In particular, his approach treats consistency rules (analogous to policy properties) as black-box entities and observes their behavior during their evaluation to detect what model elements (analogous to policy elements) they access. Then his approach decides what consistency rules to evaluate when some model elements are changed in the model. Although his approach targets at instance checking instead of assessing the quality of properties, the underlying mechanism seems similar to the preceding idea. Unfortunately, this idea has two major issues. First, it heavily depends on the way that a verification tool conducts verification. For example, a policy verification tool may access more policy elements such as rules than needed in order to verify a property due to the tool's un-optimized implementation. Second, in policy verification, verifying whether a simple property is satisfied may need to access all or most of the policy elements such as rules. Considering this simple property to cover all or most of these policy elements is too optimistic, giving policy developers false confidence on the quality of the property.

In this paper, we propose a novel approach called Mutaver that assesses the quality of properties for a policy based on mutation verification, a counterpart of mutation testing [8] in verification. Our approach is not sensitive to the internal implementation of a policy verification tool and alleviates the issue of false confidence.

Mutation testing [8] has historically been applied to general-purpose programming languages in assessing the quality of a test suite in terms of fault-detection capability. Recently mutation testing has been applied to XACML policy specifications [27] to assess the quality of a request set (test set). In our approach, we propose *mutation verification* as a means to assess the quality of a set of properties. In addition, mutation verification determines which properties interact with which rules in a policy. This information is useful in not only determining the quality of elicited properties but also during the property elicitation process.

Given a policy, our approach automatically seeds it with faults to produce numerous mutant policies, each containing one fault. Then given a property for this policy, our approach conducts property verification on this policy (called the original policy) and each mutant policy. If a property that holds for the original policy fails to hold for the mutant policy, then the mutant is said to be killed by the property. The ratio of the number of killed mutants to the total number of mutants serves as a metric to quantify the comprehensiveness of the elicited properties.

```
 1 If role = Faculty
 2   and resource = (ExternalGrades or InternalGrades)
 3   and action = (View or Assign)
 4 Then
 5   Permit
 6 If role = Student
 7   and resource = ExternalGrades
 8   and action = Receive
 9 Then
10   Permit
```

Figure 1: Rules in an example XACML policy.

This paper makes the following main contributions:

- We propose a novel approach for assessing the quality of properties for a policy in policy verification. Within the best of our knowledge, our approach is the first one to tackle this problem in policy verification and even in general software verification. The underlying idea shall have a broader implication in developing new approaches for assessing the quality of properties for other types of software artifacts.
- We implement the proposed approach with an automatic tool that facilitates automated mutation verification of access control policies written in XACML [1].
- We present a case study on an access control policy from a real-world software system to demonstrate the feasibility of this approach.

The rest of the paper is organized as follows. Section 2 presents an example to illustrate the high-level idea of the approach, Section 3 describes the background information for our mutation verification approach. Section 4 presents the mutation verification approach. Section 5 describes our experiences of applying mutation verification on a real-world policy and Section 6 discusses issues in the approach. Finally Section 7 presents related work and Section 8 concludes.

2. EXAMPLE

This section illustrates our approach to mutation verification through a simple example. The example and corresponding properties come from an example used by Fisler et al. [10]. This access control policy formalizes a university's policy on assigning and accessing grades. It is a role-based access control policy with two roles: FACULTY and STUDENT, two resources: INTERNALGRADES and EXTERNALGRADES, and three actions: ASSIGN, VIEW, and RECEIVE. For this example, we expect the following properties to hold:

Pr_1 There do not exist members of STUDENT who can ASSIGN EXTERNALGRADES.

Pr_2 All members of FACULTY can ASSIGN both INTERNALGRADES and EXTERNALGRADES.

Pr_3 There exists no combination of roles such that a user with those roles can both RECEIVE and ASSIGN the resource EXTERNALGRADES.

Property Pr_1 is intuitive since we certainly do not want students to assign grades. Property Pr_2 is to ensure that indeed faculty members can assign grades (otherwise who would assign them?). Finally, Pr_3 is an example of separation-of-duty since we do not want anyone to assign their own grade, an apparent conflict of interest.

```
 1 If role = Faculty
 2   and resource = (ExternalGrades or InternalGrades)
 3   and action = (View or Assign)
 4 Then
 5   Deny
 6 If role = Student
 7   and resource = ExternalGrades
 8   and action = Receive
 9 Then
10   Permit
```

Figure 2: The first mutant XACML policy.

```
 1 If role = Faculty
 2   and resource = (ExternalGrades or InternalGrades)
 3   and action = (View or Assign)
 4 Then
 5   Permit
 6 If role = Student
 7   and resource = ExternalGrades
 8   and action = Receive
 9 Then
10   Deny
```

Figure 3: The second mutant XACML policy.

Figure 1 shows the example XACML policy. To keep the example readable and concise, the policy is written as simple IF-THEN statements. This representation over-simplifies the complexity of XACML policies but suffices for illustrative purposes (more background information on XACML is described in Section 3). The policy in Figure 1 does not immediately satisfy these three properties. In order for the properties to hold, the policy must first be constrained. Specifically the "resource" and "action" attributes are constrained to contain single values only. Furthermore, the "subject" attribute is constrained in a slightly different manner by what Fisler et al. [10] describe as *constrain-policy-disjoint*. It is essentially a separation-of-duty constraint that restricts any request from having both the FACULTY and STUDENT roles. After constraining the policy in Figure 1, all three properties hold.

The first step of mutation verification is to generate mutant policies. We use only one of the mutation operators proposed in our previous work [27], namely *Change Rule Effect (CRE)*. The CRE mutation operator simply states to invert each rule's EFFECT by changing PERMIT to DENY or DENY to PERMIT (one at a time for each mutant policy). The number of mutant policies created by this operator is equal to the number of rules in the policy. This operator should never create equivalent mutants, which are mutant policies with the same behavior as the original policy[1], unless a rule is unreachable. The example policy has only two rules and thus only two mutant policies are generated. Figures 2 and 3 show these two mutant policies.

The second step of mutation verification is to determine which properties hold for the original policy and each mutant policy. The mutant is said to be killed by a property if the property holds for the original policy but does not hold for the mutant policy. In other words, the property reveals the fault seeded in the mutant policy. Similar to mutation testing, the greater the number of mutants killed, the more comprehensive the properties are in covering policy behaviors, and thus the more effective the properties are at interacting with the rules in the policy.

As already stated, with a properly constrained policy, the original policy satisfies all three properties; therefore, if any property does not hold for a mutant policy, then that mutant policy is killed by the property.

[1]In other words, there exists no request to make an equivalent mutant policy and the original policy to make different policy decisions.

```
1  Counterexample:
2  1:/Action, command, Receive/
3  2:/Action, command, OTHER/
4  3:/Resource, resource-class, ExternalGrades/
5  4:/Resource, resource-class, OTHER/
6  5:/Subject, role, Student/
7  6:/Subject, role, OTHER/
8  7:/Action, command, View/
9  8:/Action, command, Assign/
10 9:/Resource, resource-class, InternalGrades/
11 10:/Subject, role, Faculty/
12          1
13 1234567890
14 {
15 00000-0111
16 00100-0101
17 }
```

Figure 4: Counterexamples for the mutant policy in Figure 2.

The first mutant policy in Figure 2 does not satisfy Pr_2 and thus the first mutant is killed. Recall Pr_2 seeks to ensure that all faculty members can assign grades. Since the fault in Figure 2 is precisely the rule that grants this access, the property is apparently violated. Figure 4 illustrates the output from the property verification on the first mutant policy. Each counterexample (i.e., request) is represented as a bit mask where each bit corresponds to the specific attribute-id on Lines 2–11. If the bit is 0, then the corresponding attribute value is not present whereas if the bit is 1 then the corresponding attribute value is present. As expected, the given concrete counterexamples are for a FACULTY to ASSIGN INTERNALGRADES and for a FACULTY to ASSIGN EXTERNALGRADES. These two counterexamples correspond to Lines 15 and 16 in Figure 4, respectively. Access is denied for both requests, indicating a violation of property Pr_2.

The second mutant policy in Figure 3 is not killed by any of the three properties, reflecting that the properties are not comprehensive and do not completely "cover" the policy. The mutant coverage (i.e., the mutant-killing ratio) for the given policy by the given properties is computed as 50% since only one of two mutants is killed. This realization leads to the elicitation of our fourth property, which was not originally specified by Fisler et al. [10]:

Pr_4 All members of STUDENT can RECEIVE EXTERNALGRADES.

Property Pr_4 fails to hold for the second mutant policy in Figure 3, thus killing the mutant, revealing its fault, and increasing the mutant-killing ratio to 100%. Mutation verification serves two purposes: (1) to quantify how thorough a set of properties interacts with or covers the rules defined in the policy and (2) to facilitate property elicitation such that a property set interacts with or covers all rules defined in the policy.

3. BACKGROUND

This section presents background information including a description of XACML, policy mutation testing, and Margrave, a policy verification tool used in our approach.

3.1 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. We implement our approach to mutation verification in XACML.

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.

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 [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.

Figure 5 shows an example XACML policy, which is revised and simplified from a sample Fedora[2] policy. 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. This policy has one policy element which in turn contains two rules. The rule composition function is “first-applicable”, meaning the first applicable rule encountered during evaluation is returned as the decision. Lines $2-13$ defines the target of the policy, which indicates that this policy only applies to those access requests of an object “demo:5”. The target of Rule 1 (Lines $15-25$) further narrows the scope of applicable requests to those asking to perform action “Dissemination” on object “demo:5”. Its condition (Lines $26-35$) indicates that if the subject’s “loginId” is “testuser1”, “testuser2”, or “fedoraAdmin”, then the request should be denied. Otherwise, according to Rule 2 (Line 37) and the rule composition function of the policy (Line 1), a request applicable to the policy should be permitted. We implement our mutation verification approach for XACML access control policies.

3.2 Policy Mutation Testing

Policy mutation testing is used to measure the fault-detection capability of a request set [27]. Mutation testing [8] 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. [12] 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.

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

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

Figure 5: An example XACML policy

In order to measure the fault-detection capability of a request set, our previous work [27] developed an automated policy mutation testing approach. Given a policy, a mutator generates a number of mutant policies. Given a request set, this approach evaluates 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 the approach determines 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 [36]. 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* [36, 41] later refined into *selective mutation* [31, 32, 36] 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 [31, 32]. *Mutant sampling*, first proposed by Acree [3] and Budd [5], involves randomly selecting a subset of mutant programs which are then evaluated. Results from Wong [40] show that a 10% random sample of mutants is only 16% less effective than a full set in ascertaining fault-detection capability. Another sampling approach 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 [37].

Various test smarter approaches involve optimizations for specific computer architectures [6, 21, 29, 34] and techniques that exploit the classic space-time trade-off [11]. For example, *weak mutation* [16] 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 [15, 30, 42], studied empirically [13, 23, 35], and probed with variants that differ on exactly when the program states should be compared [30, 42]. Weak mutation has been shown to generate tests that were almost as effective as test generated with strong mutation and that at least 50% or more of the execution time was saved [33, 35].

Lots of work has been done to help overcome the expenses and barriers associated with mutation testing of general-purpose programming languages. Fortunately, policy mutation testing is not as expensive as classical mutation testing simply because policy specification languages are far simpler than general-purpose programming languages. Similarly, formal verification of policy specification are less costly. This distinction is one of the primary reasons mutation verification is feasible. Formal methods for general-purpose programming languages can be computationally expensive. The space and time cost of verification on a large number of mutant programs quickly renders mutation verification of general-purpose programming languages impractical. We use a variant of the policy mutation testing framework developed in our previous work [27] to facilitate the implementation of our mutation verification approach presented in Section 4.

3.3 Margrave

We leverage an existing verification tool called Margrave [10, 14] that consumes a policy and property and determines whether the policy satisfies the property. Margrave is a software tool suite written in PLT Scheme for analyzing access control policies written in XACML. In addition to providing a PLT Scheme API for defining and verifying properties, Margrave also performs change-impact analysis between two versions of a policy, and allows the specification of environment constraints [10]. Environment constraints are analogous to the environment models used in model checking, which bound the behaviors of the system by explicating details of the operating context in which the model will execute. In practice, to perform property verification on a policy using Margrave, a Scheme program is written that leverages the Margrave API to (1) load the policy, (2) optionally specify environment constraints on the policy, and (3) define the set of properties that the policy must satisfy.

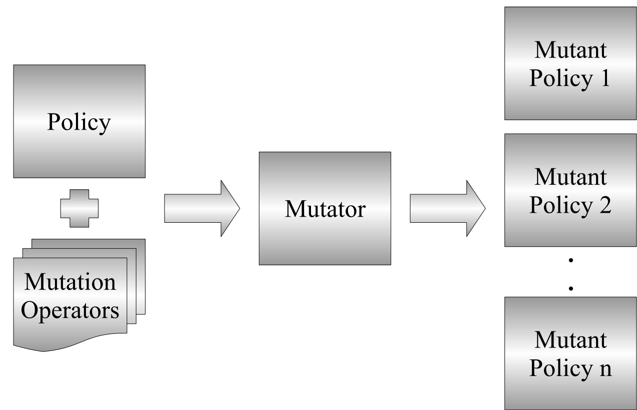

Figure 6: Mutant generation.

4. MUTATION VERIFICATION

This section presents our approach for policy mutation verification to assess the quality of policy properties. We next describe the details of each step in the approach: mutant generation, property verification, and mutant-killing determination.

4.1 Mutant Generation

Given a policy, the first step is to generate a set of mutant policies. Our previous work [27] presents a fault model for access control policies and a mutation testing framework to investigate the fault model. The framework includes mutation operators used to implement the fault model, mutant generation, equivalent-mutant detection, and mutant-killing determination. The mutant generation component [27] leverages Sun's XACML implementation [2] to iteratively manipulate an in-memory model of the XACML policy and serialize its XML representation out to disk. Previously we used mutation testing to measure the quality of a request set in terms of fault-detection capability. In our new approach, we use the mutant generation component to generate mutants based on a single mutation operator, namely *Change Rule Effect (CRE)*. We use the generated mutants not to measure the quality of a request set, but to measure the quality of a set of properties used for property verification.

Figure 6 illustrates the necessary inputs and resultant outputs of the mutant generation. The inputs are the policy under test and, in this case, a single mutation operator. The mutator then generates a set of mutant policies, each with a single fault. The CRE mutation operator generates a mutant for each rule in the policy. The mutant for a rule is generated by negating the decision of that rule. Other mutation operators have been implemented [27] but for this implementation of mutation verification we have restricted the mutator to CRE for several reasons. Mutation operators that manipulate rule conditions and the combining algorithms of POLICYSETS, POLICIES, and RULES are excluded because Margrave does not support all standard XACML combining algorithms and many condition functions. Although property verification executes relatively quickly, large policies can be used to easily generate thousands of mutant policies. We restrict ourselves to CRE not only to reduce the number of generated mutant policies but because CRE should never create equivalent mutants. An equivalent mutant is a mutant that is syntactically different from the original policy while being semantically equivalent. In other words, an equivalent mutant will produce the same result as the original policy for all inputs and thus provides no benefit, either for classical mutation testing or mutation verification. As a result, equivalent mutants cannot be

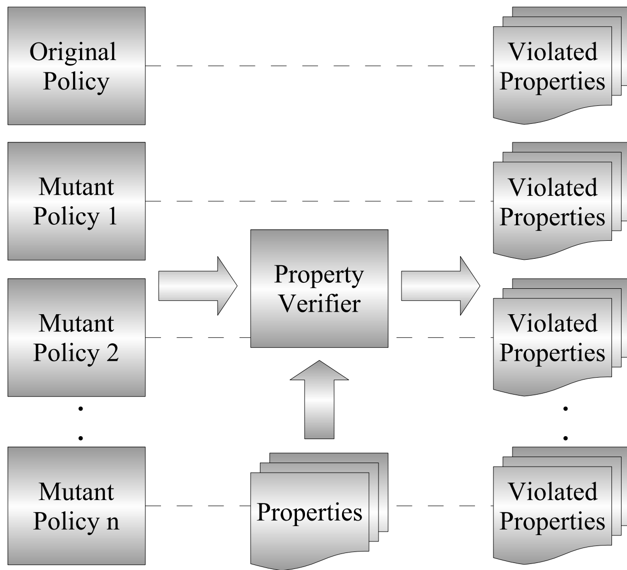

Figure 7: Property verification.

killed and would thus result in an artificial lowering of the mutant-killing ratio, giving an underrated and inaccurate quality measure for the set of properties.

4.2 Property Verification

Given a policy, a set of properties, and a set of mutant policies, the next step is to determine which properties hold and which properties do not hold for both the original policy and each mutant policy as illustrated in Figure 7. We leverage an existing policy verification tool called Margrave [10, 14] to perform property verification. Margrave is a PLT Scheme API for analyzing access control policies. Margrave represents XACML policies as multi-terminal binary decision diagrams (MTBDDs). MTBDDs are a type of decision diagram that maps bit vectors over a set of variables to a finite set of results. Margrave is implemented on top of the CUDD package [39]. CUDD provides an efficient implementation of MTBDDs. In addition to property verification, Margrave also provides semantic differencing information between version of policies [10].

To perform property verification on a policy using Margrave, a Scheme program is written that leverages the Margrave API. This program must load the policy, optionally specify any environment constraints, and define the set of properties that the policy must satisfy. In order to perform property verification programmatically, we develop an executable script and Scheme program for the original policy and each mutant policy. The script and program generation and output processing are implemented on top of tooling from the Eclipse Modeling Framework (EMF[3]) Project and the Model To Text (M2T[4]) Project. EMF is a modeling framework and code generation facility for defining a model specification and generating a set of Java classes that implement that model specification. We specified and generated an EMF model that encapsulates the necessary information to generate the executable scripts and Scheme programs. Given the file directory containing the mutant policies, we programmatically create an instance of this EMF model, which is then used as input to a JET transformation. The JET component of M2T is typically used in the implementation of a code generator for model-driven development. We use JET and a set of corresponding Java Emitter Templates to create executable shell scripts that essentially pipe a generated Scheme program to a command-line interpreter. The output of the Scheme interpreter is then piped to a trace file for further processing. These trace files contain the necessary information for determining which properties hold and which properties do not hold for the original policy and each mutant policy.

[3] `http://www.eclipse.org/modeling/emf/`
[4] `http://www.eclipse.org/modeling/m2t/`

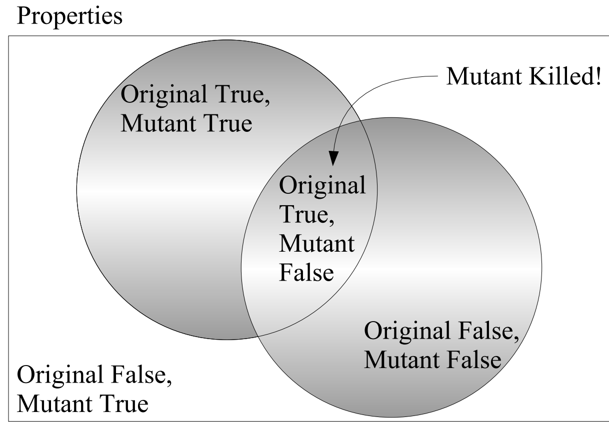

Figure 8: Venn diagram illustrating the four property states.

4.3 Mutant-Killing Determination

Finally, the next step is to compute the mutant-killing ratio. The mutant-killing ratio is the ratio of the number of mutants killed to the total number of mutants. This ratio serves as a metric to quantify the coverage of a given policy by a set of properties. A high mutant-killing ratio indicates the property set interacts with or covers a high number of rules defined in the policy.

The trace files generated by the property verification described earlier are parsed in order to divide the property set into four subsets for each mutant. A Venn diagram is illustrated in Figure 8 that describes the relationship of these four sets for a single mutant policy. The area inside the box represents the set of all properties. The area inside the left-most circle represents the set of properties that hold true for the original policy. Thus the area outside the left-most circle and inside the box is the set of properties that do not hold true for the original policy (i.e., these properties fail to be satisfied by the original policy). The area inside the right-most circle represents the set of properties that hold false for the mutant policy. Therefore, the area outside the right-most circle and inside the box represents the set of properties that hold true for the mutant policy. The area of interest is the intersection of the two circles. If at least one property holds true for the original policy but fails to hold true for the mutant policy, then the mutant is killed. If the two circles do not intersect (i.e., there are no properties that satisfy this condition), then the mutant is not killed. A property that holds true for the original policy and the mutant policy has no value in exposing the fault in the mutant policy because the property does not apply to the portion of the policy that contains the fault. A property that holds false for the original policy has no value because it is unclear if this false property is caused by an error in the policy or the property itself. More specifically, before mutation verification is conducted, these properties must be manually inspected to determine whether they fail due to an error in the policy, an error in the property, or an error in the environment constraints.

Table 1: Policies used in the case-study.

Subject	# PolicySet	# Policy	# Rule	# Property
CONTINUE-A	111	266	298	9
CONTINUE-B	111	266	306	9
SIMPLE-POLICY	1	2	2	3

Table 2: Mutant-killing ratios.

Subject	mutant-kill ratio	# mutants	# killed
CONTINUE-A	24.16%	298	72
CONTINUE-B	24.84%	306	76
SIMPLE-POLICY	50.00%	2	1

5. CASE STUDY

We have applied our mutation verification tool to an access control policy for CONTINUE [22]. CONTINUE is a web-based conference manager that supports the submission, review, discussion, and notification phases of conferences. The CONTINUE policy was used as a case study to explore property verification and change-impact analysis for Margrave by Fisler et al. [10]. The conference management system itself has been used to manage several conferences. Table 1 lists the policies used in our case study. Each row corresponds to a policy and Columns 2, 3, and 4 denote the number of POLICYSET, POLICY, and RULE elements in each policy, respectively. Column 5 denotes the number of properties used for each policy. The SIMPLE-POLICY was presented in Section 2 and CONTINUE-A and CONTINUE-B are two versions of the CONTINUE policy. All three policies and property sets are available at the Margrave web site[5].

Table 2 shows the results of mutation verification, specifically the number of mutants (Column 3), number of killed mutants (Column 4), and the mutant-killing ratio (Column 2) for each policy. As discussed in Section 2, the SIMPLE-POLICY has only two rules and thus two mutant policies. One mutant is killed so the mutant-killing ratio is simply $\frac{1}{2}$ or 50%. The complexity of the CONTINUE policies make them far more interesting. Each version of CONTINUE has approximately 300 rules and roughly $\frac{1}{4}$ of them are killed. This result is not quite surprising considering the number of rules compared to the number of properties.

To further visualize and discuss the results, let each property and each mutant be identified by an integer number. For example, let the original policy be denoted P_0, each mutant policy be denoted $P_1, P_2, \ldots P_m$, and each property $Pr_0, Pr_1, \ldots Pr_{p-1}$ where m and p are the number of mutants and properties, respectively. A policy-property pair (P_i, Pr_j) is mapped to a point (i, j) in Figures 9 and 10. A data point is plotted on the chart at (i, j) if the property Pr_j fails to hold for Policy P_i. Therefore, Figures 9 and 10 illustrate all property failures for each policy-property pair. More specifically, each integer value along the x-axis denotes a single property and each integer value along the y-axis denotes a single policy. Furthermore, the policy at $y = 0$ is the original (unmutated) policy. These scatter plots allow us to quickly determine which properties interact with which rules in the policy.

Property Pr_0 fails to hold for the first version of CONTINUE (P_0 in Figure 9) and thus also fails to hold for any mutant policies as indicated by the numerous data points along $x = 0$. The natural language for this property is as follows:

Pr_0 If the subject is a pc-member, it is not the discussion phase,

[5] http://www.cs.brown.edu/research/plt/software/margrave/

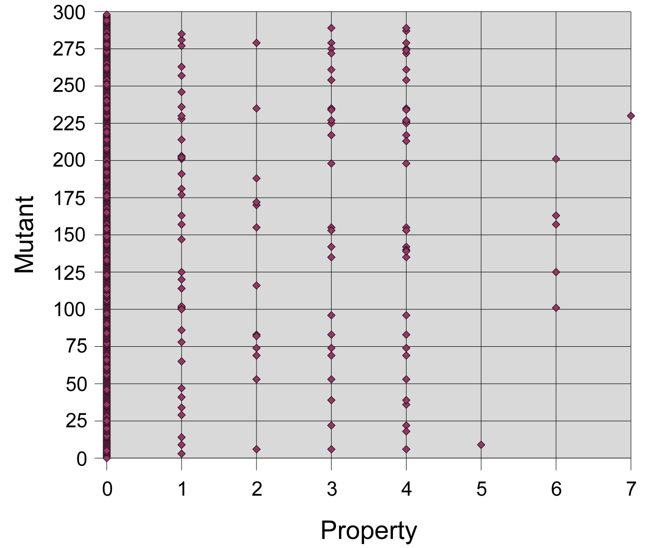

Figure 9: CONTINUE-A property failures for each policy-property pair.

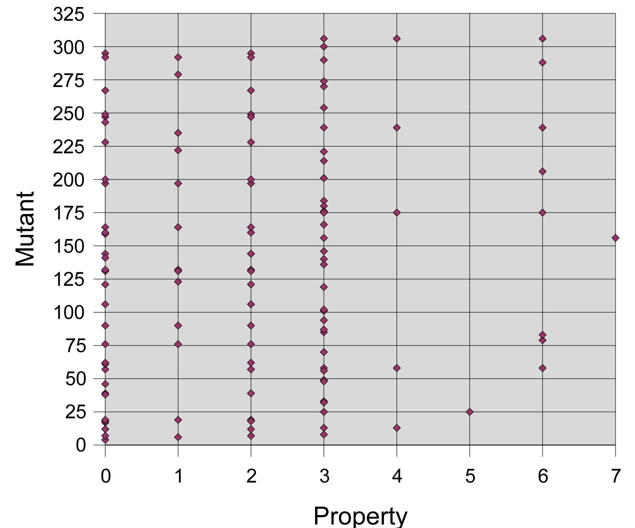

Figure 10: CONTINUE-B property failures for each policy-property pair.

and unsubmitted for the review for a paper despite being assigned it, then the subject cannot see all parts of other's reviews for that paper.

This property fails simply because CONTINUE-A is an earlier version of the policy. All properties including Pr_0 do hold for the revised version in Figure 10. Another readily noticeable peculiarity of Figure 9 is the absence of Pr_8. Recall that nine properties are verified against each policy implying one property, Pr_8, does not appear to interact with any rule explicitly defined in the policy. The natural language for the "missing" property is:

Pr_8 No legal request is mapped to Not Applicable, that is every legal request is decided by either deny or permit.

Pr_8 is an excellent example of a valid property that is not explicitly specified in the policy itself. A policy should certainly be written such that every legal request returns a deny or permit response. This property, however, is a generic property potentially applicable to a wide range of policies. Although the property is quite relevant, it is not (and arguably should not) be specified explicitly in the policy itself. An argument against its inclusion in the policy itself is that the property is generic; in particular, it is unrelated to the access control logic of the system but is rather a best practice. This type of generic property is not accounted for in this implementation of mutation verification. Further investigation is needed to determine how to incorporate such properties. For instance, mutation operators that consider not only the policy but also the properties may account for these types of properties.

The second version of CONTINUE (P_0 in Figure 10) satisfies all properties as indicated by the lack of data points along $y = 0$. Again, Pr_8 (i.e., $x = 8$) is not plotted because this generic property does not interact directly with any rules specified in the policy. Properties Pr_5 and Pr_7 are interesting because they fail for only a single rule for both versions of CONTINUE. The natural language for these properties are:

Pr_5 If a subject is not a pc-chair or admin, then he may not set the meeting flag.

Pr_7 If some one is not a pc-chair or admin, then he can never see paper-review-rc for which he is conflicted.

By manual inspection, we determine the mutant killed by Pr_5 is the same for both versions of the policy. The killed mutant corresponds to the last rule in the POLICYSET that specifies access to the meeting flag. More specifically, once all permitted combinations of subjects and actions are specified, the final rule ensures all other requests for the meeting flag are denied. Because the mutant policy changed this rule's decision to permit, the mutant was killed by Pr_5. In a similar fashion, the killed mutant for Pr_7 is identical for both versions and corresponds to precisely the rule that ensures the denial of requests for paper reviews when the isConflicted flag is set.

The CONTINUE policy heavily uses the first-applicable combining algorithm. As a result, it is often the case that, for a given resource, all permitting requests are specified first followed by a more general denying request. When these types of denying rules are mutated to permit, the policy leaks sensitive information (i.e., access is granted when it should not). A general property for ensuring that sensitive information remains protected is effective at identifying these leaks. For example, properties Pr_1 in Figure 9 and Pr_3 in Figure 10 are in fact the same property. This property interacts with a large number of policy rules indicated by the large number of data points. This property in natural language states that if the subject role attribute is empty and the resource class is not conference info, then return deny. This property effectively identifies information leakage introduced through the mechanism described earlier. This result indicates that this property set is effective at identifying information leakage in the policy.

On the other hand, the mutants that are not killed are generally those that mutate a permitting rule to deny. For example, when the rule that allows the admin to read the pcMember-info-isChairFlag is switched from permit to deny, no property identifies the restricted access. Similar to having general properties for ensuring that sensitive information remains protected, you also want to have properties for ensuring access is granted when appropriate. The fact that the un-killed mutants are generally of this type indicates that the property set can be improved by adding properties for ensuring that access is granted when appropriate.

6. DISCUSSION

Our approach to mutation verification provides a coverage measure of a policy by a set of properties. If a property set achieves a mutant-killing ratio of 100%, can we say the property set is exhaustive or complete? This situation is similar to statement coverage in software testing. If a test suite achieves 100% statement coverage for a given program, can we say the test suite can find all defects in the program? The answer, of course, is absolutely not. While mutation verification serves as a quality measure for a property set and, with the current mutation operator, identifies which properties interact with which rules in the policy, it may not consider more abstract, generic properties. For example, Pr_1 of the illustrative example in Section 2 ensures a student cannot assign grades. While this property is an intuitive one of the problem domain, it is not explicitly expressed in the policy itself. This particular policy contains only rules that *allow* access whereas this property is concerned with *denying* access. The fact that this property does not interact with the rules in the policy does not imply it is not needed. A better example is discussed in Section 5 where the property serves as more of a best practice that is not related to the problem domain of the access control.

Further exploration of mutation operators for mutation verification is needed to investigate how to reflect relevant properties (that are not necessarily specified in the policy itself) in the mutation verification process. Despite this shortcoming, our investigation supports the feasibility of mutation verification for large, complex policies. Mutation verification provides a coverage metric for a policy relative to a property set and can identify weak areas of the properties that should be supplemented with additional properties.

7. RELATED WORK

To help ensure the correctness of policy specifications, researchers and practitioners have developed formal verification tools for policies. Several policy verification tools are developed specifically for firewall policies. Al-Shaer and Hamed [4] developed the Firewall Policy Advisor to classify and detect policy anomalies. Yuan et al. [43] developed the FIREMAN tool to detect misconfiguration of firewall policies.

There are also several verification tools available for `XACML` policies [1]. Hughes and Bultan [17] translated `XACML` policies to the Alloy language [18], and checked their properties using the Alloy Analyzer. Schaad and Moffett also leverage Alloy to check that role-based access-control policies do not allow roles to be assigned to users in ways that violate separation-of-duty constraints [38]. Zhang et al. [45] developed a model-checking algorithm and tool support to evaluate access-control policies written in *RW* languages,

which can be converted to XACML [44]. Kolaczek proposes to translate role-based access-control policies into Prolog for verification [19]. Kolovski et al. [20] formalize XACML policies with description logics (DL), which are a decidable fragment of first-order logic, and exploit existing DL verifiers to conduct policy verification. Fisler et al. [10] 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 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. Most of these approaches require user-specified properties to be verified. Our new approach complements these existing policy verification approaches because our approach helps assess the quality of the properties during policy verification.

Our previous work proposed an approach to policy property inference via machine learning [25]. Such properties are often not available in practice and their elicitation is a challenging and tedious task. Furthermore, once properties are defined, it is difficult to measure their effectiveness and identify potential problems areas that need improvement. Our mutation verification framework intends to help alleviate that challenge. Our implementation leverages Margrave's property verification feature to to verify properties against mutant policies.

Although various coverage criteria [46] for software programs exist, only recently have coverage criteria for access control policies been proposed [28]. 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. Our previous work [28] 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, several test generation techniques have been developed. The first one iterates over all possible requests for a given policy, if its domain set is finite [28]. The second one is a random test generation tool that randomly generates tests for XACML policies [28]. The third technique [26] is a novel framework that automatically generates high-quality tests based on a change-impact analysis tool such as Margrave [10]. Different from these policy testing approaches, our new approach focuses on assessing the quality of properties in policy verification.

To our knowledge, no metric has yet been defined to quantify the coverage of a policy by some property set. Our previous work [27] defined a fault model and corresponding automated mutator in order to quickly evaluate test generators and techniques of test selection in terms of fault-detection capability. We leverage a variation of this automated mutator in our implementation of the mutation verification framework.

8. CONCLUSION

The need for carefully controlling access to sensitive information is increasing as the amount and availability of data is growing. In order to separate the semantics of access control from the distributed system itself, access control policies are increasingly specified in domain-specific, declarative languages such as `XACML`. Doing so facilitates managing, maintaining, and analyzing of policies. To increase confidence in the correctness of specified policies, policy authors can formally verify policies against a set of properties. Policy verification is an important technique for high assurance of the correct specification of access control policies. Since the effectiveness of the verification process is directly related to the quality of the properties, we have proposed a novel approach to assess the quality of a set of properties. We have presented an approach to mutation verification of access control policies and a tool that implements the approach on `XACML` policies. Similar to the way mutation testing is used to measure the quality of a test suite in terms of fault-detection capability, mutation verification is used to measure the quality of a set of properties. In other words, mutation verification allows us to quantify the coverage of a given policy by a property set. Given a policy and a set of properties, our approach generates several mutant policies, each with a single fault. Then our approach verifies the property set against the original policy and each mutant policy. The property set is then partitioned into four subsets for each mutant policy in order to compute the mutant-killing ratio. We applied our mutation verification tool to policies and properties from a real-world software application. Our experiences show that the performance of the property verification is encouraging and mutation verification can scale to sufficiently large access control policies. Furthermore, mutation verification is a complementary approach to property verification by aiding in the elicitation of properties.

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] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proc. 23rd Conf. IEEE Communications Soc. (INFOCOM 2004)*, 2004.

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

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

[7] 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.

[8] 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.

[9] A. Egyed. Instant consistency checking for the UML. In *Proc. 28th International Conference on Software Engineering*, pages 381–390, 2006.

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

[11] V. N. Fleyshgakker and s. N. Weiss. Efficient mutation analysis: A new approach. In *Proceedings of the International Symposium on Software Testing and Analysis*, 1994.

[12] 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.

[13] M. R. Girgis and M. R. Woddward. An integrated system for program testing using weak mutation and data flow analysis. In *Proceedings of the 8th International Conference on Software Engineering*, 1985.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] G. Kolaczek. Specification and verification of constraints in role based access control for enterprise security system. In *WETICE '03: Proc. the 12th International Workshop on Enabling Technologies*, page 190, Washington, DC, USA, 2003. IEEE Computer Society.

[20] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*, May 2007.

[21] E. W. Krauser, a. P. Mathur, and V. Rego. High performance testing on simd machines. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, 1988.

[22] S. Krishnamurthi. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, January 2003.

[23] B. Marick. The weak mutation hypothesis. In *Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification*, 1991.

[24] E. Martin and T. Xie. Automated test generation for access control policies. In *Supp. Proc. 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, November 2006.

[25] E. Martin and T. Xie. Inferring access-control policy properties via machine learning. In *Proc. 7th IEEE Workshop on Policies for Distributed Systems and Networks (POLICY 2006)*, pages 235–238, June 2006.

[26] 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 (SESS 2007)*, pages 5–11, May 2007.

[27] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proc. 11th International Conference on World Wide Web (WWW 2007)*, May 2007.

[28] 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 (ICICS 2006)*, pages 139–158, December 2006.

[29] 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.

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

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

[32] 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.

[33] A. J. Offutt and S. D. Lee. How strong is weak mutation? In *Proceedings of the 4th Symposium on Software Testing, aNalysis, and Verification*, 1991.

[34] A. J. Offutt, R. Pargas, S. V. Fichter, and P. Khambekar. Mutation testing of software using a mimd computer. In *International Conference on Parallel Processing*, 1992.

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

[36] 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.

[37] M. Sahinoglu and E. H. Spafford. A bayes sequential statistical procedure for approving software products. In *Proceedings of the IFIP Conference on Approving Software Products*, 1990.

[38] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *ACM SACMAT 2002 Symposium on Access Control Models and Technologies*, 2002.

[39] F. Somenzi. CUDD: CU Decision Diagram Package Release, 1998.

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

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

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

[43] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: A toolkit for FIREwall Modeling and ANalysis. In *Proceedings of 2006 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[44] 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.

[45] 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.

[46] 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.