Testing Access Control Policies

JeeHyun Hwang[1] Evan Martin[1] Tao Xie[1] Vincent C. Hu[2]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206

[2] Computer Security Division, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930

{jhwang4,eemartin}@ncsu.edu xie@csc.ncsu.edu vhu@nist.gov

Abstract

As software systems become more and more complex, and are deployed to manage a large amount of sensitive information and resources, specifying and managing correct access control policies is critical and yet challenging. Policy testing is an important means to increasing confidence in the correctness of specified policies and their implementations for access control. There are two types of policy testing. In the first type, the artifacts under test are policy specifications and the main testing goal is to assure the correctness of the policy specifications. In the second type, the artifacts under test are policy implementations and the main testing goal is to assure the conformance between the policy specifications and implementations. Both types of policy testing supply typical test inputs (requests) to the artifacts under test and subsequently check test outputs (responses) against expected ones. This article presents recent approaches on policy testing in five main categories: fault models, testing criteria, test generation, test oracles, and model-based testing.

Keywords: Policy Testing, Security Policies, Access Control Policies, Access Control Models, Coverage Criteria, Test Generation, Test Oracles

1 Introduction

Access control is one of the most fundamental and widely used privacy and security mechanisms for a system to share information in dynamic and distributed environments. Access control mechanisms control which principals such as users or processes have access to which resources in a system. Access control policies (or policies for simplicity) can be specified in programming languages or policy specification languages and retrofitted or implemented in a particular access control implementation. Policies need to be carefully designed and implemented to prevent data from unauthorized access. Correctly specifying policies is crucial because correct implementation and enforcement of policies by systems are based on the premise that the policy specifications are correct.

However, specifying and managing access control policies are not trivial; it is common that a system's privacy and security are compromised due to the misconfiguration of access control policies instead of the failure of cryptographic primitives or protocols. The problem becomes increasingly severe as software systems become more and more complex, and are deployed to manage a large amount of sensitive information and resources that are organized into sophisticated structures.

A policy implementation handles an access request and determines if an access to a resource is permitted based on the specified policies. Correct policy implementations of policy specifications are not an easy task. For example, in distributed systems, a variety of multiple policy implementations must be integrated to control administration, users, databases, and various services. Most of the policy decision functionalities are distributed and may include dynamic policy decisions or different domain-specific policy implementations. Policy implementations can be implemented in various ways (explicitly or implicitly) such as configurable components and program code. In a legacy system, policy implementations may adopt domain-specific access control mechanisms. In such a system, policy implementations may include policies in program code as being entangled with other functionalities. This entangled code is difficult to analyze, modify, and test for the policies in code. Thus, when policy requirements change, the developers may need to modify the entangled functionalities to comply with the change, and it is not trivial to identify and adapt the corresponding program code. Moreover, policy implementations may include security vulnerabilities: the developers may seed malicious code (such as "backdoors" and malfunctions) in a system, or the developers may misrepresent or forget important access authorization enforcements. Therefore, both policy specifications and implementations must undergo rigorous verification and validation through systematic testing to ensure that the policy specifications and implementations truly encapsulate the desires of the policy authors.

Policy testing is the testing process to assure the correctness of policy specifications and implementations. With adequate policy testing, one can increase confidence on the correctness of policy specifications and implementations. By observing the execution of a policy implementation with a test input (i.e., access request), the testers may identify any faults in the policy specification or implementation, and validate whether the corresponding output (i.e., access decision) is intended. Moreover, potential malfunctions or missing control in the policy specification or implementation can be identified during the test execution. Although policy testing mechanisms vary because there is no single standard way to specify or implement access control policies, in general the main goals to conduct policy testing are as follows: assure the correctness of the policy specifications and assure the conformance between the policy specifications and implementations.

The rest of the article is organized as follows. Section 2 explains access control concepts and policy specifications to provide background information for policy testing. This section also describes a simple XACML [1] policy in terms of its syntactic and semantic structure. Section 3 describes policy testing concepts by comparing policy testing with policy verification and traditional software testing. Section 4 presents policy testing approaches in five main categories: fault models, testing criteria, test generation, test oracles, and model-based testing. Section 5 concludes.

2 Access Control Policies

In this section, we first introduce access control policies and their relations with other access control concepts. We next describe policy specification and a simple XACML [1] policy to illustrate the structure, syntax, and semantics of a policy specification language.

2.1 Access Control Concepts

There are four main concepts in access control: access control systems, policies, models, and mechanisms [6]. Figure 1 shows the relations among these four concepts. Access control systems consist of two main levels of abstractions: (1) specification of access control policies, which are enforced by (2) implementation of access control mechanisms. *Access control policies* are high-level requirements for specifying access control. *Access control mechanisms* translate an access request to an access decision based on the access control policies. Examples of access control policies include three main types: model-based policies, rule-based policies, and the mixture of the two types. *Model-based policies* instantiate well-known access control models such as Role-Based Access Control (RBAC) [4]. *Access control models* are usually written to describe properties of access control. Therefore, model-based policies can be verified against the properties of the underlying access control models to detect property violations [7]. Users see access control models as an unambiguous and precise expression of requirements; vendors and system developers see access control models as design and implementation requirements. Without instantiating access control models, *rule-based policies* are based on pre-determined and configured rules. Unlike model-based policies defined based on well-known models, there is no commonly understood definition or formally defined standard for rule-based policies, and rule-based policies are commonly referred to a wide range of policies in some form of predefined rules. Rule-based policies can be combined with model-based policies to form the third main type of policies: the mixture of model-based and rule-based policies.

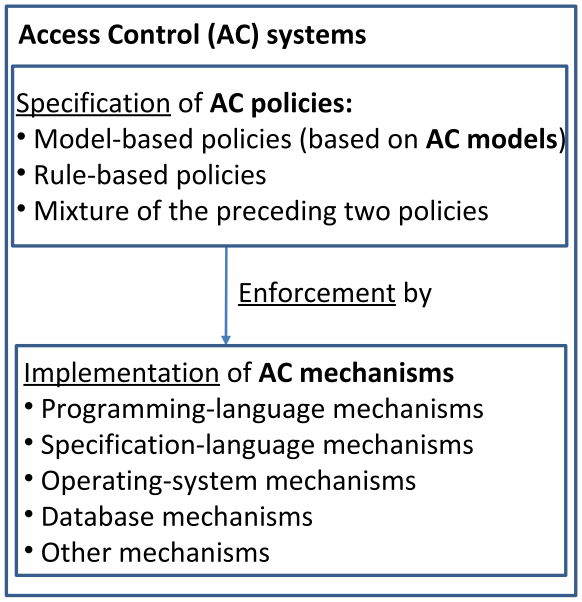
Access Control (AC) systems
Specification of AC policies:
• Model-based policies (based on AC models)
• Rule-based policies
• Mixture of the preceding two policies
Enforcement by
Implementation of AC mechanisms
• Programming-language mechanisms
• Specification-language mechanisms
• Operating-system mechanisms
• Database mechanisms
• Other mechanisms

Figure 1. Relations among access control (AC) systems, policies, models, and mechanisms.

Access control policies can be enforced by policy implementations via implementation of access control mechanisms. Typical types of access control mechanisms include (1) policies written in programming languages, and embedded and enforced in software systems, (2) policies written in policy specification languages such as XACML [1] and enforced through a Policy Decision Point (PDP), (3) policies embedded and enforced in operating systems (such as policies enforced to protect files and directories), and (4) policies embedded and enforced in database systems (such as policies enforced to regulate access to tables and views).

2.2 Policy Specification

Various policy specification languages can be used to express a policy with regards to the intent of organizations and users. Policy specification languages can help the policy authors to manage policies in various stages (such as writing, reviewing, analyzing, testing, and deploying policies). However, there is no dominant way to specify policies and each specification language has its own syntax, semantics, and features.

It is not trivial to implement policy implementations correctly from specifications written in programming languages or formal logic-based specification languages. In contrast to programming languages, a formal logic-based specification language represents logical and precise fine-grained expressions of policies. However, it is also not trivial for the policy authors to encode complex logic in policy expressions.

There is no universal or automated way to implement policy specifications. Policy implementations may be implemented by domain experts in various ways (explicitly or implicitly) such as configurable components and program code. In the process of implementing a policy implementation (according to a given specification), manual effort may introduce security vulnerabilities (in the implementation level) that do not exist in the specification level. Moreover, in a complex system, security aspects (reflecting policy behavior) can be implemented as being entangled with other functional aspects. In such a system, it is challenging to identify and reason about such security aspects by analyzing program code itself.

2.3 XACML Policy Specification

Recently platform-independent policy specification languages such as XACML [1] have been increasingly used in various applications. In these languages, policies are defined separately from a Policy Decision Point (PDP), which can be treated as a policy evaluation engine, and other functional aspects of a system. In other words, a policy specification and a PDP are treated as a static configuration and a policy evaluation engine, respectively. This feature makes it easy for the policy authors to manage policies because policies (in such platform-independent languages) can often be directly modified and applied without changing the PDP implementation. However, if the policy authors need customized or new features (e.g., dynamic constraints) that are not supported by the PDP implementation, its modification is needed.

XACML (eXtensible Access Control Markup Language) [1] is a platform-independent XML-based policy specification language proposed by OASIS. By design, XACML is a rule-based specification language; however, various profiles of XACML have been proposed to specify model-based policies such as RBAC [4]. An XACML policy specification consists of a policy set and a policy combining algorithm. A *policy set* is an ordered list of policies. A *policy* includes a target, a rule set, and a rule combining algorithm. A *target* is a predicate over the subject, the resource, and the action of access requests, specifying the type of requests to which the policy can be applied.

```
<Policy Id="univ" RuleCombAlgId="first-applicable">
<Target>
  <Subjects> <AnySubjects/> </Subjects>
  <Resources><AnyResources/> </Resources>
  <Actions> <AnyActions/> </Actions>
</Target>
<Rule RuleId="1" Effect="Permit">
 <Target>
  <Subjects><Subject> Faculty </Subject></Subjects>
  <Resources> Grades </Resources>
  <Actions><Action> Write </Action>
           <Action> View </Action></Actions>
 </Target></Rule>
<Rule RuleId="2" Effect="Deny">
 <Target>
  <Subjects><Subject> Student </Subject></Subjects>
  <Resources>Grades </Resources>
  <Actions><Action> Write </Action></Actions>
 </Target>
</Rule>
</policy>
```

Figure 2. An example XACML policy.

If a request satisfies the target of a policy, then the request is further checked against the rule set of the policy; otherwise, the policy is skipped without further examining its rules. A *rule set* is an ordered list of rules. A *rule* consists of a target, a condition, and an effect. The *target* of a rule is similar to the target of a policy, but specifies whether the rule is applicable to a request. Given a request, if a rule is applicable, the *condition* (i.e., a boolean function) associated with the rule is evaluated. If the condition is evaluated to be true, the rule's *effect* (i.e., `Permit` or `Deny`) is returned as a *decision*; otherwise, `NotApplicable` is returned as a decision. If an error occurs when a request is applied against policies or their rules, `Indeterminate` is returned as a decision.

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

In summary, in XACML, the policy authors specify *policies*, *rules*, *targets*, and *conditions* to describe the policy behavior. Although some syntax and structures of XACML may not be common across other policy specification languages, XACML can be flexibly used to specify various types of policies (e.g., RBAC and rule-based policies) based on specific XACML profiles.

Figure 2 shows an example XACML policy. This example illustrates a policy that uses the first-applicable algorithm, which determines to return the evaluated decision of the first applicable rule. In this example, there are two subjects (i.e., roles) (`Faculty`, `Student`), one resource (`Grades`), and two actions (`View`, `Write`). There are two rules in the policy. Lines 7-14 define the first (permit) rule, specifying that a faculty member can write or view grades. Lines 15-21 define the second (deny) rule, specifying that a student cannot write grades.

3 Policy Testing Concepts

This section describes policy testing concepts by comparing policy testing with other closely related (but different) concepts such as policy verification and software testing.

To increase confidence in the correctness of specified policies, the policy authors can conduct policy testing by supplying typical test inputs (requests) and subsequently checking test outputs (responses) against expected ones. A test case refers to a test input equipped with its expected output. Through dynamic evaluation of test inputs, the policy authors may detect faults in either policy specifications or implementations.

Through policy testing, one can detect whether there are faults in a policy specification or implementation. For example, to check whether a certain property (that can be encapsulated into requests) of the policy is violated, one can generate and evaluate the corresponding requests against the policy. Therefore, one can validate if the specified policy truly captures the desired policy behavior.

3.1 Comparison of Policy Verification and Testing

While policy testing is dynamic analysis of access control policies through dynamic request evaluation, policy verification is static analysis, which does not require dynamic request evaluation. Policy verification is an important means to ensuring the correct specification of access control policies [8, 18] but it faces several issues that pose barriers for being widely adopted in practice. First, it is often difficult for policy authors to come up with properties [5] (required by policy verification) beyond the policy rules written by the policy authors. In policies, formal properties are often not available and the policy authors would need to come up with properties by "thinking out of the box", which

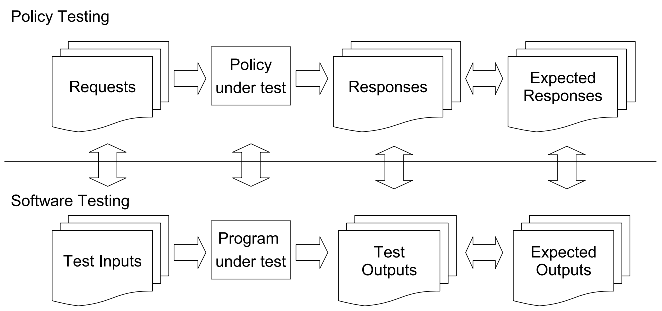

Figure 3. Analogy between policy testing and software testing.

is often challenging, because the policy authors would often view that the written policy rules already encode properties reflecting the intended policy behavior in their mind. Second, policy verification tools cannot support a full feature set of access control policies and these unsupported features may be often used in real policies in practice. For example, Margrave [5], an XACML policy verification tool, does not support condition elements found within a rule element or hierarchical data. Verifying these features would make the verification problem intractable. Third, while a policy specification is correct, its policy implementation may be faulty and produce a wrong decision. For example, the policy-implementation developers may implement malicious code (such as "backdoors") in a system. These faults cannot be detected by policy verification but can be detected by policy testing. In addition, within the supported features, it is often expensive to verify complex access control policies in large-scale systems.

3.2 Comparison of Software Testing and Policy Testing

Software testing is an important and practical approach to efficiently detecting faults in complex software systems. Faults in policy specifications may also be detected by leveraging and adapting existing techniques for software testing. Figure 3 illustrates the analogy between policy testing and software 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 (i.e., the policy evaluation engine) against the access control policy under test. Policy testers can inspect request-response pairs to check whether they are as expected[1].

[1]The results (responses) of policy testing may be affected if the underlying PDP implementation contains faults, although general PDP implementations are often well tested ahead of time. To help isolate faults in a PDP implementation, policy testers can run the failing tests on an-

On the other hand, there are differences between software testing and policy testing. Policy testing focuses on ensuring correct access control behaviors and direct application of software testing (such as functional testing) techniques may not be well suitable for the following reasons. First, access control policies are often described in declarative languages such as XACML and are rule-based, without explicit control flows as defined in programming languages. Second, a fault in a policy usually has smaller impact on the behavior and is more difficult to detect than a fault in software programs, partly because policy outputs or behaviors fall into mainly a limited-element domain such as `Permit` and `Deny`. Therefore, observing responses may not help expose a fault when the faulty rule or implementation portion is evaluated but the response is correct. Third, there are unique features in policies that can be particularly fault-prone. Such example features include rule orders and role hierarchies in RBAC. To detect faults related to these features, the design of policy testing techniques should take policy domain knowledge into account.

4 Policy Testing Approaches

This section describes policy testing approaches in five main categories: fault models, testing criteria, test generation, test oracles, and model-based testing. These categories of policy testing approaches are closely related. Fault models provide insights into the types of policy faults, providing a foundation for policy testing. Testing criteria provide guidelines for test generation in terms of what test inputs to generate and help reduce manual effort in constructing test oracles by reducing generated test inputs. Model-based testing represents a typical approach in policy testing.

4.1 Fault Models

A basic fault model describes small syntactic and semantic policy faults. An advanced fault model describes more sophisticated policy faults (e.g., those that are seeded by malicious policy authors or implementers).

4.1.1 Basic Fault Models

There exist various basic fault models [14, 16, 17, 19] for different types of policies. Martin and Xie [14] proposed a fault model to describe simple faults in XACML policies. They categorized faults broadly as syntactic and semantic faults. Syntactic faults are a result of simple typos whereas semantic faults are involved with the logical constructs of the policy language.

other different PDP implementation to see whether the failure can be reproduced [10].

Table 1. Index of mutation operators.

Type	ID	Description
Syntactic faults	PSTF	Policy Set Target False
	PTF	Policy Target False
	RTF	Rule Target False
	RCF	Rule Condition False
Semantic faults	CPC	Change Policy Combining Algorithm
	CRC	Change Rule Combining Algorithm
	CPO	Change Policy Order
	CRO	Change Rule Order
	CRE	Change Rule Effect
	PSTT	Policy Set Target True
	PTT	Policy Target True
	RTT	Rule Target True
	RCT	Rule Condition True

Based on the fault model, mutation operators describe modification rules for modifying access control policies to seed faults in policies as listed in Table 1. We next describe the concepts of these syntactic faults and semantic faults (with their related mutation operators) in XACML policies.

Syntactic Faults. Syntactic faults consist of simple typos that are easy to make, and result in a syntactically faulty policy. Basic static analysis tools exist to check for such syntactic faults. For example, in XACML, an XML schema definition (XSD) can be used to check for obvious syntactic faults. However, syntactic faults that do not violate the XSD can occur due to typos in attribute values. For example consider Line 9 of the example XACML policy in Figure 2. The attribute value “Faculty” can accidentally contain a typo (e.g. “Faulty”) causing the target to be incorrectly specified.

Three mutation operators PSTF, PTF, and RTF that emulate syntactic faults are proposed [14]. These three mutation operators ensure that a target evaluates to false for policy sets, policies, and rules, respectively. These operators correspond to typos found in the target such as the preceding example that results in the target not applying to any (correctly specified) access requests. Typos can also exist in a condition and can result in the condition always evaluating to false, which can be emulated by the RCF mutation operator.

Semantic Faults. Semantic faults are more elusive than syntactic faults and involve incorrect use of logical constructs of the policy specification language. For XACML policies, these logical constructs include policy or rule combining algorithms, policy evaluation order, rule evaluation order, and various functions found in the condition. Because these are logical faults in the construction of the policy, it is unlikely that static analysis can detect such faults.

Several mutation operators [14] emulate semantic faults. Two mutation operators CPC and CRC mutate the policy and rule combining algorithms, respectively. CPO and CRO mutate the policy and rule orders, respectively, which may be a common semantic fault when using order-sensitive combining algorithms. CRE is another semantic fault that occurs when an incorrect decision is specified. PSTT, PTT, and RTT ensure that the target evaluates to true for policy sets, policies, and rules, respectively. This case could happen if the policy author fails to define a target for each of the elements. Similarly, RCT ensures that the condition always evaluates to true, which may occur if the policy author fails to define a condition.

Mutation operators related to policies and conditions may not be applicable to other types of policies than XACML policies because the constructs of policies and conditions are not universally used in various types of policies. Some other types of access control policies may have their special syntactic and semantic constructs calling for their own mutation operators. For example, Traon et al. [19] designed mutation operators on a given Organization Based Access Control (OrBAC) [2] model. They designed similar semantic and syntactic mutation operators as the preceding ones in addition to other mutation operators including rule deletion, rule addition, and role change based on a role hierarchy. Masood et al. [16, 17] designed a fault model assuming that potential faults are located in the policy implementation. They did not design specific mutation operators based on the fault model; instead, they described incorrect implementations (and their effects) in states, edges, and paths in the Finite State Machine (FSM) model for representing policies such that if states and transitions (denoted as edges in the model) are implemented incorrectly, then implemented policy behavior will not conform with the FSM model.

4.1.2 Advanced Fault Models

Advanced fault models also exist to emulate complicated policy faults [16, 17]. In practice, a fault can be more complicated than involving just one simple mutation. A malicious policy author or implementor may seed some difficult-to-detect faults that cause the implementation (denoted as PI) to behave incorrectly under specific conditions. Therefore, the policy testers need to generate test cases to detect such faults that allow unauthorized access. Advanced fault models characterize complex faults including counter-based and I/O-based faults [16, 17] as follows.

Counter-based faults. PI counts certain conditions/activities during execution. PI starts to behave incorrectly when the count is greater than a given number k. For example, if `Bob` is assigned to the role `Banker` more than 100 times a day, `Bob` can access unauthorized data.

I/O-based faults. Upon receiving a specific input, PI starts to behave incorrectly. For example, a malicious user can be allowed to access unauthorized data if she/he enters a special keyword (that seems to be random) such as "8ab3e2a3aaud".

These types of faults are serious because PI evaluates requests incorrectly and these types of faults are difficult to be detected by normal test cases.

4.2 Testing Criteria

Testing criteria help determine whether sufficient testing has been conducted and it can be stopped, and measure the degree of adequacy or sufficiency of a test suite (consisting of test cases). Among testing criteria for policy testing, structural coverage criteria are defined based on observing whether each individual policy element has been evaluated when a test suite (request set) is evaluated by a PDP. Fault coverage criteria are defined based on observing whether each (seeded) potential fault is detected by the test suite. Fault coverage is commonly measured with mutation testing techniques.

4.2.1 Structural Coverage Criteria

In traditional software testing, code coverage is a metric that reports which parts of program code have been covered (tested) by the test cases. By observing code coverage, testers or test generation tools may generate test cases to cover not-yet-covered code parts or stop testing if the desired code coverage is reached. While a policy has different constructs from program code, structural coverage criteria specific to the policy domain can be designed to measure the degree to which parts of a policy has been tested through evaluating test cases with a PDP.

Martin et al. [15] proposed structural coverage criteria for XACML policies based on observing whether each individual policy element is involved when a request is evaluated. If no requests are evaluated against a rule during testing, then potential faults in that rule cannot be exposed. Thus, it is important to generate requests so that a large portion of rules are involved in the evaluation of at least one of the requests. The higher the coverage percentage, the higher quality the request set is. In XACML, there are three major types of policy elements: policies, rules for each policy, and a condition for each rule. Then three policy coverage metrics [15] for these policy elements are defined as below:

- *Policy coverage.* A policy is covered by a request if the policy is applicable to the request *and* the policy contributes to the decision. That is to say, the request satisfies all the conditions in the policy's target and the PDP has yet to fully resolve the decision for the given request. Policy coverage is measured as 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. That is to say, the policy is applicable to the request and the request satisfies the conditions in the rule's target, and the PDP has yet to fully resolve the decision for the given request. Rule coverage is measured as the number of covered rules divided by the number of total rules.

- *Condition coverage.* The condition for a rule can be evaluated to true or false. These two outcomes are called 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 measured as the number of covered true conditions and covered false conditions divided by twice of the number of total conditions.

The defined structural coverage criteria are domain-specific: the criteria are designed based on XACML's syntax and its structure to express a policy. While the criteria are not intended to be universally applicable for other types of policies, some coverage types in the criteria may be applicable to other types of policies. For example, rule coverage criteria could be generally applicable in various types of rule-based policies. However, the policy and condition coverage criteria may not be applicable to other types of policies because XACML used the terms of conditions and policies (with its own definition) to express policies and their constraints. Some other types of access control policies may have their own special constructs; their own coverage criteria types need to be defined to accommodate detecting potential faults introduced on these special constructs.

For example, Traon et al. [19] developed an approach that defines rule coverage criteria for OrBAC policies [2]. Their approach defines primary and secondary rule coverage criteria. In OrBAC, one element can be classified to sub-elements. For example, the `Person` role (called a super-role) can be classified to sub-roles (the `Manager` role and the `Employee` role). Through a role hierarchy, sub-roles inherit a super-role's permission or prohibition. Their approach defines primary and secondary rule coverage for different types of rules being evaluated (with regards to the type of roles in a rule — super-roles or sub-roles). Masood et al. [16, 17] proposed structural coverage criteria for policies represented as an FSM.

4.2.2 Fault Coverage Criteria

Fault coverage criteria are defined to measure fault-detection capability of a test suite. Let us consider two test suites, namely T_1 and T_2. If T_1 detects more faults than T_2, we consider that the quality of T_1 is better than T_2 in terms of fault coverage. Generally fault coverage is measured by seeding faults in the policy or its implementation under test via mutation testing (described in Section 4.2.3).

Martin and Xie [14] measured fault coverage by seeding faults in the XACML policy under test. Their fault detection is defined based on observing whether the response produced by a PDP against a faulty policy is different from the one produced by the same PDP against the original policy when a request is evaluated. Their approach seeded syntactic or semantic faults in an XACML policy while assuming that the XACML PDP is already well-tested and may not contain faults.

Masood et al. [16, 17] measured fault coverage by seeding faults in the policy implementation under test. Their criteria are in the context of conformance testing with an FSM for representing policies, and their fault detection is defined based on observing whether the result produced by the faulty implementation is different from the one produced by the original implementation when a test case is executed. In their FSM, to detect seeded faults in the implementation, it is important to generate test cases to cover each state, edge, or path where potential faults may be located. Moreover, they also seeded complicated faults that may not be detected easily. One of such complicated faults is a counter-based fault: a fault is detected only after executing some states or edges for a certain number of times. Detecting such faults calls for sophisticated test generation techniques.

Strong correlations could exist between fault coverage criteria and structural coverage criteria. Intuitively, there is a higher chance to detect faults if more elements in policies are covered (thus increasing the chance of covering the elements that include the faults). Generally when the types and locations of seeded faults are comprehensive enough, fault coverage criteria shall be stronger or more difficult to satisfy than structural coverage criteria. However, comprehensive comparison among various policy testing criteria is yet to be studied both analytically and empirically.

4.2.3 Mutation Testing

Policy mutation testing has been commonly used to measure fault coverage in terms of satisfying fault coverage criteria [14, 16, 17, 19]. Mutation testing [3] has been historically applied to programming languages to help measure fault coverage. Mutation testing involves mutants that include modified program code (in small ways) different from the original program code. Because some faults exist in

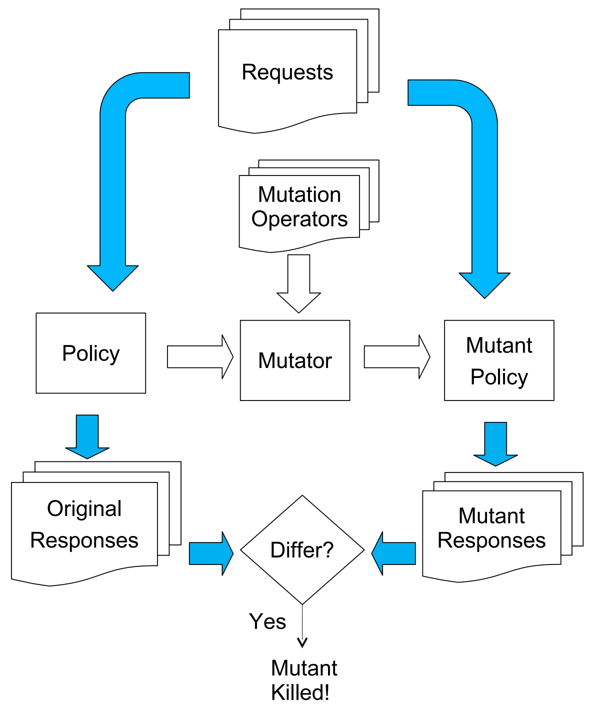

Figure 4. Overview of policy mutation testing.

mutants, the testers can check whether some mutants are killed (i.e., detected) through observing executions on both the original program and its mutants.

The program under test, test inputs, and test outputs in software testing correspond to the policy, requests, and responses in policy mutation testing, respectively. An overview of policy mutation testing is illustrated in Figure 4. The key component of policy mutation testing is a set of mutation operators, whose details are described in Table 1. Given a policy and a set of mutation operators, a mutator generates a number of mutant policies. The policy under test is iteratively mutated according to mutation operators (developed based on fault types in fault models described in Section 4.1) to produce numerous mutants, each containing one fault. Each request in the given request set is evaluated 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 mutant policy is determined to be killed by the request; otherwise, the mutant policy is not killed.

Mutation testing involves mutants that are modified from the original policy syntactically or semantically using mutation operators. However, mutation operators are domain-specific. Different types of policies may have their special constructs, and their mutation testing may require the design of their domain-specific mutation operators.

The fundamental premise of mutation testing 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 case that also detects that fault. In other words, the capability to detect small, minor faults such as mutants implies the capability to detect complex faults. The higher the percentage of killed mutants, the more effective the test suite is at fault detection or fault coverage.

4.3 Test Generation

To test access control policies, policy testers can manually generate test inputs (i.e., requests) to achieve high structural policy coverage and fault coverage (i.e., fault-detection capability). To reduce manual effort, automated test generation approaches can be used. We next describe three main types of automated test generation approaches for policy testing: random test generation, test generation based on change-impact analysis, and test generation for model-based policies.

Random Test Generation. For XACML policies, Martin and Xie [12] developed an approach for random test generation. The approach analyzes the policy under test and generates requests by randomly selecting requests from the set of all possible requests. The analysis cost for generating each request is very low but the randomly generated requests may not achieve high structural coverage or fault coverage. Random test generation is considered as a shallow approach because some important requests for corner cases have a low probability of being generated compared to a systemic test generation approach as described below.

Test Generation based on Change-Impact Analysis. Martin and Xie [12] also developed a test generation approach that generates requests by iteratively manipulating inputs to Margrave [5], which can conduct change-impact analysis. Given the policy version under test, the approach automatically synthesize a new policy version (in a similar way as policy mutation described in Section 4.2.3) such that its semantic differences from the given policy version are specific coverage targets (e.g., a not-yet-covered rule). Then the given policy version and its synthesized policy version are fed to Margrave, which produces counterexamples (as generated requests) to witness the semantic differences of the two policies. Change-impact analysis can help generate requests to cover potentially changed parts across versions (e.g., parts that are not yet covered by the existing requests). However, change-impact analysis is often expensive and is inefficient for test generation if two policy versions (for change-impact analysis) are lengthy and highly different semantically.

Test Generation for Model-based Policies. Test generation for model-based policies [9, 11, 16, 17, 19] derives abstract test cases (directly from models such as FSMs that specify model-based policies) using certain testing criteria to search only desired requests. Note that abstract test cases may not be executable and they need to be refined to concrete test cases in order to be executed. Masood et al. [17] generated test cases to cover a complete FSM for RBAC policies; therefore, the generated test cases could help detect faults in an edge or state of the FSM. Moreover, they could generate test cases to cover all paths from the FSM's root entry to all states and leaves to detect faults in such paths. Traon et al. [19] proposed test generation techniques to cover rules specified in policies based on the OrBAC [2] model. They first identified rules from the policy specification and generated test cases to be applicable to a certain rule. Section 4.5 gives more details on general techniques of model-based testing.

These existing test generation approaches are centered around covering rules and other domain-specific constructs. These approaches are generally applicable to other types of policies. The quality of the generated requests can be assessed with the testing criteria described in Section 4.2.

4.4 Test Oracles

After requests (i.e., test inputs) are generated via a test generation approach, the policy or implementation under test is used to evaluate each generated request. The request evaluation produces an actual response, which is compared against the expected response. If the actual response is the same as the expected response, the request is classified as a *passing* request; otherwise, the request is classified as a *failing* request. A test oracle is a mechanism to determine the expected response. There are two main approaches for test oracles in policy testing: (1) manually inspecting actual responses and (2) using properties to check actual responses.

Manual Inspection. Policy testers can inspect the responses of requests and decide whether the responses are correct based on the desired policy behavior reflected by requirements. However, if there are a large number of requests being generated, such manual effort is not feasible, being time-consuming, tedious, and error-prone activities. To reduce such effort, Martin and Xie [15] used structural coverage criteria (described in Section 4.2) to select a subset of requests for response inspection such that the selected subset achieves the same structural coverage as the original request set. Such request-set reduction could reduce inspection effort while incurring low loss in fault-detection capability (i.e., fault coverage). Applying heuristics on the FSM for RBAC policies to focus on detecting a certain type of faults, Masood et al. [17] selected a subset of test cases (for inspection) from the complete set of test cases.

Property Checking. Policy testers can first formulate properties for policies and then use them to check the responses for requests. When such properties are available, policy testers can construct assertions to automatically check the correctness of the responses of requests against the properties. Masood et al. [17] derived a policy's properties directly from its specification or model, which is assumed correct. Then they automatically checked the responses of requests against the properties to detect faults. However, it is often challenging for the policy authors to come up with properties. To alleviate the issues, Martin and Xie [13] developed an approach to infer likely properties. The approach analyzes the pairs of request-response under test and uses machine learning techniques to infer likely properties.

4.5 Model-based Testing

Model-based testing derives test cases from a model[2] (such as an FSM) that represents truly desired behaviors of the policy or policy implementation under test. Generally, the directly derived test cases are abstract and not executable on the policy implementation. These abstract test cases should be refined to concrete test cases suitable for execution, and should hold the same objective of the corresponding abstract test cases. For example, if an abstract test case is intended to help evaluate a certain rule, the execution of the corresponding concrete test case should evaluate the same rule. Then the concrete test cases are executed against the policy implementation to check whether any faulty behavior is observed. If faulty behavior is observed, the policy tester should inspect whether there are any specification or implementation faults against the given model.

There are three main reasons that model-based testing is adopted for policy testing. (1) A model represents correct policy behavior. (2) Policies can be enforced in implementations in different ways and it is expensive to derive test cases from each concrete implementation. (3) If we directly derive test cases from a system that includes a policy implementation, many test cases may test functional aspects (instead of security aspects).

There exist various approaches on model-based policy testing. Masood et al. [16,17] modeled an RBAC [4] policy as an FSM model. The FSM model represents two major activities in RBAC: user-role assignments and user-role activations. For example, in a company there are two roles (`Manager` and `Employee`), two users (`Mary` and `Jim`), and a resource (`salary`). To access the `salary`, `Mary` first needs to be assigned to `Employee` (which is inactive at the moment), and then the corresponding role `Employee` should be activated. Note that only authorized users can be assigned to roles and only when a role is active, the associated users can then take an action (such as checking her

[2]The model here is different from the access control model described in Section 2. The model here can be considered as a type of policy specifications. But model-based policies (policies based on access control models) are often amenable to being specified with a model here.

`salary`) related to the role. An FSM consists of states that represent all possible states of user-role assignments and user-role activations. An edge in an FSM denotes a transition between states. From an FSM, their approach generates tests cases to cover specific states, edges, or paths.

Traon et al. [19, 20] developed an approach to test a policy-based library management system, which adopts the OrBAC [2] model. From the system requirements, they identified explicit rules related to access control. The requirements also include constraints that the maximum number of books (that a user can borrow) varies by roles such as public user or student roles. Their approach can visualize rules in OrBAC and detect conflicting rules. Based on the model, their approach can also generate abstract test cases to cover rules.

Li et al. [9] and Mallouli et al. [11] developed approaches to generate abstract test sequences automatically from Extended Finite State Machines (EFSM) models and define testing criteria. In their approaches, EFSM models and testing criteria can model OrBAC policies and types of rules (such as permission, prohibition, or obligation rules), respectively. One can generate and select suitable test sequences (to satisfy the testing criteria) from available paths in the exploration of EFSM models.

There are two issues to be addressed when adapting traditional model-based testing to policy testing. First, appropriate models need to be derived from policy requirements. When policy requirements are unavailable, model-based testing may not be applicable because models needed by model-based testing are not available. Second, testing criteria need to be defined. Testing criteria help identify and select desirable test cases among all possible test cases. With testing criteria, the number of selected test cases can be effectively reduced.

Model-based testing approaches can be applied to conduct conformance testing. Conformance testing is to check whether there are faults in a policy specification or its implementation. In model-based testing, with the mapping between a model and its implementation, Masood et al. [16, 17] and Traon et al. [19, 20] proposed to conduct model-based conformance testing with additional steps (after generation of abstract test cases): (1) generation of concrete test cases (from abstract test cases), (2) execution of the concrete test cases, and (3) comparison of the expected and actual results of the concrete test cases. Other work [9, 11] showed preliminary results on generating test cases from models in conducting conformance testing. Model-based conformance testing can often be practical in policy testing because (abstract) test cases are generated from a model independently of its actual implementation (which is often domain-specific and can be derived in various ways).

5 Conclusion

Specifying and managing correct access control policies are critical and yet challenging. Policy testing is an important means to increasing confidence in the correctness of a specified policy and its implementation. Policy testing has its own distinct features different from policy verification and traditional software testing. We have presented policy testing approaches in five closely related main categories: fault models, testing criteria, test generation, test oracles, and model-based testing.

Acknowledgment

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

References

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

[2] A. Abou El Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization based access control. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, pages 120–131, 2003.

[3] 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, 1978.

[4] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

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

[6] V. C. Hu, D. F. Ferraiolo, and D. R. Kuhn. Assessment of access control systems. Interagency Report 7316, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, September 2006.

[7] V. C. Hu, E. Martin, J. Hwang, and T. Xie. Conformance checking of access control policies specified in XACML. In *Proceedings of the 1st IEEE International Workshop on Security in Software Engineering (IWSSE 2007)*, pages 275–280, July 2007.

[8] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2004)*, pages 31–42, 1997.

[9] K. Li, L. Mounier, and R. Groz. Test generation from security policies specified in Or-BAC. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, pages 255–260, 2007.

[10] N. Li, J. Hwang, and T. Xie. Multiple-implementation testing for XACML implementations. In *Proceedings of the Workshop on Testing, Analysis and Verification of Web Software (TAV-WEB 2008)*, pages 27–33, 2008.

[11] W. Mallouli, J.-M. Orset, A. Cavalli, N. Cuppens, and F. Cuppens. A formal approach for testing security rules. In *Proceedings of the 12th ACM symposium on Access Control Models and Technologies (SACMAT 2007)*, pages 127–132, 2007.

[12] E. Martin and T. Xie. Automated test generation for access control policies. In *Supplemental Proceedings of the 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, 2006.

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

[14] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*, pages 667–676, 2007.

[15] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proceedings of the 8th International Conference on Information and Communications Security (ICICS 2006)*, pages 139–158, 2006.

[16] A. Masood, A. Ghafoor, and A. Mathur. Scalable and effective test generation for access control systems. Technical Report SERC-TR-285, Purdue University Electrical and Computer Engineering Department, 2005.

[17] A. Masood, A. Ghafoor, and A. Mathur. Test generation for access control systems that employ RBAC policies. Technical Report SERC-TR-283, Purdue University Electrical and Computer Engineering Department, 2005.

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

[19] Y. L. Traon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. In *Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE 2007)*, pages 93–102, 2007.

[20] Y. L. Traon, T. Mouelhi, A. Pretschner, and B. Baudry. Test-driven assessment of access control in legacy applications. In *Proceedings of the 1st International Conference on Software Testing Verification and Validation (ICST 2008)*, pages 238–247, 2008.