Automated Coverage-Based Testing of XACML Policies

Dianxiang Xu
Department of Computer Science
Boise State University
Boise, ID 83725, USA
dianxiangxu@boisestate.edu

Roshan Shrestha
Department of Computer Science
Boise State University
Boise, ID 83725, USA
roshanshrestha@boisestate.edu

Ning Shen
Department of Computer Science
Boise State University
Boise, ID 83725, USA
ningshen@boisestate.edu

ABSTRACT

While[1] the standard language XACML is very expressive for specifying fine-grained access control policies, defects can get into XACML policies for various reasons, such as misunderstanding of access control requirements, omissions, and coding errors. These defects may result in unauthorized accesses, escalation of privileges, and denial of service. Therefore, quality assurance of XACML policies for real-world information systems has become an important issue. To address this issue, this paper presents a family of coverage criteria for XACML policies, such as rule coverage, rule pair coverage, decision coverage, and Modified Condition/Decision Coverage (MC/DC). To demonstrate the assurance levels of these coverage criteria, we have developed methods for automatically generating tests, i.e., access requests, to satisfy the coverage criteria using a constraint solver. We have evaluated these methods through mutation analysis of various policies with different levels of complexity. The experiment results have shown that the rule coverage is far from adequate for revealing the majority of defects in XACML policies, and that both MC/DC and decision coverage tests have outperformed the existing methods for testing XACML policies. In particular, MC/DC tests achieve a very high level of quality assurance of XACML policies.

KEYWORDS

XACML, access control, coverage criteria, test generation, mutation testing

ACM Reference Format:

Dianxiang Xu, Roshan Shrestha, and Ning Shen. 2018. Automated Coverage-Based Testing of XACML Policies. In *SACMAT'18*: 23rd ACM Symposium on Access Control Models & Technologies, June 13–15, 2018, Indianapolis, IN, USA. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3205977.3205979

1 INTRODUCTION

XACML (eXtensible Access Control Markup Language) [1] is an OASIS standard for specifying attribute-based access control (ABAC) policies in the XML format. ABAC [2] is a new access control method where authorization elements are defined in terms of attributes, rather than identities, of subjects, actions, resources, and environments. These attributes are characteristics of subjects (e.g., job title and age), actions, resources (e.g., data, programs, and networks), and environments (e.g., current time and IP address) that are predefined and pre-assigned by an authority [3]. By combining various attributes into access control decisions, ABAC enables fine-grained access control of resources. ABAC also facilitates collaborative policy administration within a large enterprise or across multiple organizations [1]. The Federal Identity, Credential, and Access Management (FICAM) Roadmap and Implementation Plan v2.0 [4] has called out ABAC as a recommended access control model for promoting information sharing between diverse and disparate organizations. The National Strategy for Information Sharing and Safeguarding included a Priority Objective that the federal government should extend and implement the FICAM Roadmap across federal networks in all security domains [3]. Currently XACML3.0 has been used in the mainstream identity management products, such as Oracle's Identity Manager and WSO2's Identity Server.

XACML supports a variety of data types, functions, and combining algorithms for policy composition. While such expressiveness is highly desirable for representing real-world ABAC policies, it raises challenges for validating whether XACML policies indeed meet the access control requirements. When an ABAC policy is coded in XACML, defects can be introduced for various reasons, such as misunderstanding of the access control requirements, omissions, and coding errors [5]. These defects may result in unauthorized accesses, escalation of privileges, and denial of service. To reveal these defects, a major practice is to test the policy by feeding the policy together with test inputs to an XACML engine (or policy decision point) and check if the policy interpreted by the XACML engine produces correct responses. A test input is an access request that consists of attribute names, types, and values. In this paper, we assume that the implementation of the XACML engine conforms to the XACML3.0 standard. The response to an access request by a given policy is consistent with the standard.

Several methods have been proposed to generate test inputs for XACML 1.0 or 2.0 policies [6]-[14]. These methods, however, have two problems. First, their experimental results have shown that they are incapable of detecting the majority of defects and produce a large number of tests. Second, the tests generated by these methods do not achieve adequate coverage of the XACML policy under test. For example, some of them [9] [10] do not

SACMAT'18, June 13-15, 2018, Indianapolis, IN, USA

ACM ISBN 978-1-4503-5666-4/18/06...$15.00
https://doi.org/10.1145/3205977.3205979

necessarily cover all reachable rules in the policy under test, while others [8][11] aim at selecting or generating test inputs to cover all rules. This paper will show that a test suite for rule coverage is far from adequate for quality assurance. Adequate test coverage is important because a defect in a policy element will not be revealed unless the policy element is exercised by some test. This paper demonstrates that coverage-based testing is very useful for quality assurance of XACML policies.

The contributions of this paper are as follows:

- We define a family of coverage criteria for XACML 3.0 policies, including rule coverage, decision coverage, Modified Condition/Decision Coverage (MC/DC), rule pair coverage, and permit/deny rule pair coverage. These coverage criteria are defined over the essential access control constraints in XACML 3.0 policies, including policy set target, policy target, rule target, and rule condition. They can be used to measure the adequacy of tests for an XACML policy under development and determine whether or not more tests need to be created and performed in order to achieve an expected level of quality assurance. Even for a policy in operational use (i.e., its tests are actual access requests in an operational environment), the coverage criteria can indicate the confidence levels of policy quality. For example, when the actual accesses are already MC/DC-adequate and none of them have led to security violation, we are confident that the policy is highly assured, even though it has not been tested adequately before the deployment.
- We have developed efficient methods for automatically generating tests to meet each of these coverage criteria using a constraint solver. They have been applied to a number of policies with different levels of complexity and demonstrated satisfactory time performance.
- We have conducted empirical studies to evaluate the cost-effectiveness of the coverage-based tests through comprehensive mutation analysis of XACML policies. The defect detection capability is measured by mutation score, i.e., mutant-killing ratio between the total number of policy mutants killed by a test suite and the total number of non-equivalent policy mutants. A policy mutant is a variant of the original policy with an injected fault. The injected faults represent the typical defects that may occur in XACML policies. A mutant is said to be killed if there is at least one test that reports a failure. Our experiment results have shown that both the MC/DC tests and the decision coverage tests are much more effective than the existing testing methods for XACML policies. In particular, MC/DC-adequate tests can provide high assurance of XACML policies.

The remainder of this paper is organized as follows. To make the paper self-contained, Section 2 briefly introduces XACML 3.0 policies. Section 3 defines the test coverage criteria. Section 4 describes the coverage-based test generation methods. Section 5 presents the empirical studies. Section 6 reviews related work. Section 7 concludes this paper.

2 ACCESS CONTROL POLICIES IN XACML3.0

The main components of the XACML3.0 model are rule, policy, and policy set. As the most elementary unit of policy, a rule consists of a target, a condition, and an effect. The target is a logical expression that specifies the set of requests to which the rule is intended to apply. The condition is a Boolean expression that refine the applicability of the rule established by the target. A policy comprises a policy target, a rule-combining algorithm identifier, and a list of rules. A policy set consists of a policy set target, a policy-combining algorithm identifier, and a list of policies or policy sets. The target of a rule, policy, or policy set is a conjunctive sequence of AnyOf clauses. Each AnyOf clause is a disjunctive sequence of AllOf clauses, and each AllOf clause is a conjunctive sequence of match predicates. A match predicate compares attribute values in a request with the embedded attributes. Logical expressions for match predicates and rule conditions are usually defined on four categories of attributes: subject, resource, action, and environment. They can use a great variety of predefined functions and data types.

We use the policy in Figure 1 as a running example. It is one of the sample policies in Balana, an open source implementation of XACML 3.0 [15]. For simplicity, some text is not omitted. The policy is named "KmarketBluePolicy" and the rule combining algorithm is *deny-overrides* (line 2). The policy's target (lines 3-14) means *role=blue*, where *role* is an attribute in the subject category and both *role* and *blue* are strings. There are three rules: *deny-liquor-medicine* (line 16-37), *max-drink-amount* (lines 38-61), and *permit-rule* (line 62).

The target of rule *deny-liquor-medicine* (lines 18-35) means *resource-id=Liquor* (line 19-26) $\vee$ *resource-id=Medicine* (lines 27-34), where *resource-id* is an attribute in the resource category. Because the rule's condition is omitted, the rule will result in a "deny" decision if *resource-id=Liquor* $\vee$ *resource-id=Medicine*. The target of rule *max-drink-amount* means *resource-id=Drink*, and the condition means *amount≥10*. Thus the rule results in a deny decision if *resource-id=Drink* $\wedge$ *amount≥10*. Rule *permit-rule* has neither target nor condition. It results in a permit decision whenever it is applied.

To facilitate our discussion, we use policy set as a general structure of XACML specification. Formally, a policy set *PS* is a triple $< pst, pca, [P_1, P_2,..., P_m]>$, where *pst* is the policy set target, *pca* is the policy combining algorithm, and $[P_1, P_2,..., P_m]$ is the list of policies in the policy set. $<pst, pca, [P_1, P_2,..., P_m]>$ reduces to a policy when *pst* and *pca* are omitted and *m=1*. Thus, the discussions in the subsequent sections apply to individual policies. Each policy P_i is a triple $<pt_i, rca_i, [R_1, R_2,..., R_n]>$, where pt_i is the policy target, rca_i is the rule combining algorithm, and $[R_1, ..., R_n]$ is the list of rules in the policy. Each rule R_j is a triple $< rt_j, rc_j, re_j >$, where rt_j is the rule target, rc_j is the rule condition, and $re_j \in$ {*Permit, Deny*} is the rule effect. $< rt_j, rc_j,$ *Permit*> is called a permit rule, whereas $< rt_j, rc_j,$ *Deny*> is a deny rule. If both rt_j and *rc* are omitted (always true), then the rule $< \_, \_, re_j >$ is a default rule. More specifically, < _, _, *Deny*> is a default deny rule, whereas < _, _, *Permit*> is a default permit rule.

```
1  <Policy xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
2  PolicyId="KmarketBluePolicy" RuleCombiningAlgId= "…deny-
   overrides" Version="1.0">
3  <Target>
4    <AnyOf>
5      <AllOf>
6        <Match MatchId="…function:string-equal">
7         <AttributeValue
   DataType="…#string">blue</AttributeValue>
8           <AttributeDesignator AttributeId="…role"
9           Category="…subject-category:access-subject"
10          DataType="…string" MustBePresent="true"/>
11       </Match>
12     </AllOf>
13   </AnyOf>
14 </Target>
15 …
16 <Rule Effect="Deny" RuleId="deny-liquor-medicine">
17 <Target>
18   <AnyOf>
19     <AllOf>
20       <Match MatchId="…function:string-equal">
21        <AttributeValue
   DataType="…string">Liquor</AttributeValue>
22        <AttributeDesignator AttributeId="…:resource-id"
23         Category="…attribute-category:resource"
24         DataType="…string" MustBePresent="true"/>
25       </Match>
26     </AllOf>
27     <AllOf>
28       <Match MatchId="…function:string-equal">
29      <AttributeValue
   DataType="…string">Medicine</AttributeValue>
30        <AttributeDesignator AttributeId="…resource-id"
31          Category="…attribute-category:resource"
32          DataType="…string" MustBePresent="true"/>
33       </Match>
34     </AllOf>
35   </AnyOf>
36  </Target>
37 </Rule>
38 <Rule Effect="Deny" RuleId="max-drink-amount">
39   <Target>
40     <AnyOf>
41       <AllOf>
42         <Match MatchId="…function:string-equal">
43         <AttributeValue
   DataType="…string">Drink</AttributeValue>
44           <AttributeDesignator AttributeId="…resource-id"
45           Category="…attribute-category:resource"
46           DataType="…string" MustBePresent="true"/>
47         </Match>
48       </AllOf>
49     </AnyOf>
50   </Target>
51   <Condition>
52     <Apply FunctionId="…function:integer-greater-than">
53       <Apply FunctionId="…function:integer-one-and-only">
54         <AttributeDesignator AttributeId="…amount"
55         Category="…category"
56         DataType="…#integer" MustBePresent="true"/>
57       </Apply>
58       <AttributeValue
   DataType="…integer">10</AttributeValue>
59     </Apply>
60   </Condition>
61 </Rule>
62 <Rule RuleId="permit-rule" Effect="Permit"/>
63 </Policy>
```

Figure 1: A sample XACML policy.

An access request consists of attribute names, types, and values. For an access request, a policy or policy set responds with an access decision, such as permit or deny. The semantics of a policy set $PS = < pst, pca, [P_1, P_2,..., P_m]>$ can be informally described as follows: given an access request q, PS is evaluated to produce a response (i.e., access decision) to q, denoted as $d(PS, q)$. Policy set target pst is first evaluated according to the attribute values in q. If the result of evaluation is false (or an error occurs during the evaluation), then $d(PS, q) = Not\text{-}Applicable$ (or *Indeterminate*), otherwise policies P_1, P_2,..., and P_m will be evaluated. $d(PS, q)$ depends on policy combining algorithm pca and the decisions of individual policies with respect to q (denoted as $d(P_i, q)$). Similarly, for an individual policy $P_i = <pt_i, rca_i, [R_1, R_2,..., R_n] >$, policy target pt_i is evaluated according to the attribute values in q. If the evaluation result is false (or an error occurs during the evaluation), then $d(P_i, q) = Not\text{-}Applicable$ (or *Indeterminate*), otherwise rules R_1, R_2,..., and R_n will be evaluated. $d(P_i, q)$ depends on rule combining algorithm rca and the decisions of individual rules. Decision of rule $R_j = < rt_j, rc_j, re_j >$ with respect to q, denoted as $d(R_j, q)$, is defined as follows:

- *Permit*: access is granted when $re_j = Permit$ and $rt_j \wedge rc_j$ is true with respect to q.
- *Deny*: access is denied when $re_j = Deny$, and $rt_j \wedge rc_j$ is true with respect to q.
- *NotApplicable*, or simply *N/A*: q is not applicable, i.e., $rt_j \wedge rc_j$ is false with respect to q.
- *IndeterminateD* or simply *I(D)*: An error occurred when rt_j or rc_j was evaluated and re_j=*Deny*. The decision could have evaluated to *Deny* if no error had occurred.
- *IndeterminateP*, or simply *I(P)*: An error occurred when rt_j or rc_j was evaluated and re_j=*Permit*. The decision could have evaluated to *Permit* if no error had occurred.

For a default rule $r_j = < \_, \_, re_j >$ and any access request q, $d(r_j, q) = re_j$. A syntactically valid access request may cause the occurrence of a runtime error for different reasons, such as missing an attribute value, mismatch of an attribute type, and an exception of expression and function evaluation.

In XACML 3.0, there are 11 rule combining algorithms and 12 policy combining algorithms (11 of them use the same names as respective rule combining algorithms). Four of them are for compatibility support of old versions - *Legacy Ordered-deny-overrides*, *Legacy Permit-overrides*, *Legacy Ordered-permit-overrides*, and *Legacy Ordered-permit-overrides*. In Balana [15], the implementations of *Ordered-deny-overrides* and *Ordered-permit-overrides* are the same as *Deny-overrides* and *Permit-overrides*. As such, our work focuses on five rule combining algorithms and six policy combining algorithms: *Deny-overrides*, *Permit-overrides*, *Deny-unless-permit*, *Permit-unless-deny*, *First-applicable*, and *Only-one-applicable*.

3 TEST COVERAGE CRITERIA FOR XACML

A test case for a policy set or policy is an access request (i.e., test input) together with the correct response to the request by the policy set or policy under test (i.e., oracle value). Generally oracle values for given test inputs are determined by the access

control requirements. In an evolving policy development process, the actual responses of test inputs from earlier policy versions can be recorded and then used as the oracle values for testing the current or future versions if their correctness is confirmed. A test passes (or fails) if the actual response is the same as (or different from) the oracle value. Sometimes we simply refer to a test as an access request (i.e., the input part of the test). A test suite is a set of tests. In the following, we define a family of test coverage criteria for XACML policies.

3.1 Rule Coverage (RC)

Definition 1. A test suite *TS* for a policy set *PS* is said to satisfy *Rule Coverage (RC)* of *PS* if, for each rule R_j in each policy P_i of *PS*, there is as least one test *q* in *TS* that evaluates rule R_j to its specified effect re_j i.e., $d(R_j, q) = re_j$.

A test *q* making R_j evaluate to its specified effect re_j must satisfy the following three conditions:

(1) Reachability of policy P_i: the test must reach policy P_i, otherwise no rule in P_i will be evaluated. This means that *PS*'s target evaluates to true and no policy before P_i terminates the evaluation of *PS*. For any policy P_k $(0<k<i)$ before P_i, its evaluation will make P_i unreachable if:
- $d(P_k, q) = Deny$ when *pca = Deny-overrides / Permit-unless-deny*,
- $d(P_k, q) = Permit$ when *pca = Permit-overrides / Deny-unless-permit*, or
- $d(P_k, q) \neq N/A$ when *pca = First-applicable.*

(2) Reachability of rule R_i: After P_i is reached, the test triggers the evaluation of rule R_i only if the policy target of P_i evaluates to true and no rule before R_i in P_i terminates the evaluation of P_i. For any rule R_s $(0<s<j)$ before R_j, its evaluation will make R_s unreachable if:
- $d(R_s, q) = Deny$ when rca_j *= Deny-overrides / Permit-unless-deny*,
- $d(R_s, q) = Permit$ when rca_j *= Permit-overrides / Deny-unless-permit*, or
- $d(R_s, q) \neq N/A$ when rca_j *= First-applicable.*

(3) Reachability of rule effect re_j: the test makes evaluate to its specified effect only if $rt_i \wedge rc_i$ evaluates to true.

For instance, a test that covers rule *max-drink-amount* in the running policy example must satisfy the following conditions:
- Rule reachability: *role=blue* $\wedge \neg$ *(resource-id=Liquor* $\vee$ *resource-id=Medicine)*
- Effect reachability: *resource-id=Drink* $\wedge$ *amount*$\geq$*10*

3.2 Decision Coverage (DC)

A policy set *PS* has different points of decision-making, such as policy set target, policy target, rule target, and rule condition. Each of these decision points can evaluate to true, false, or error. These different evaluation results lead to different access control decisions. It is desirable to test whether these decision points work correctly. In the following, we refer to policy set target, policy target, rule target, and rule condition collectively as ***decision expressions***.

Definition 2. A test suite *TS* for a policy set *PS* is said to satisfy *Decision Coverage (DC)* of *PS* if *TS* covers all three decisions of each decision expression, i.e.,

(1) *TS* has three tests to make policy set target *pst* evaluate to true, false, and error, respectively,

(2) For each policy P_i in policy set *PS*, *TS* has three tests to make policy P_i's target pt_i evaluate to true, false, and error, respectively, and

(3) For each rule R_j in each policy P_i of *PS*, *TS* has three tests to make rule target evaluate to true, false, and error, respectively.

(4) For each rule R_j in each policy P_i of *PS*, *TS* has three tests to make rule condition evaluate to true, false, and error, respectively.

In (1), if a test makes policy set target *pst* evaluate to true, then individual policies in *PS* will continue to be evaluated. If a test makes *pst* evaluate to false or error, then the evaluation of *PS* result in a decision of *N/A* or *Indeterminate*. A test that makes *pst* evaluate to error (called error test) refers to a valid access request that leads to the *Indeterminate* decision due to such semantic issues as missing attribute value and mismatch of attribute type. This is similar for policy targets, rule targets, and rule conditions. Section 3.5 will discuss why error tests are useful for detecting defects in XACML policies. As described before, the reachability condition of P_i is assumed in (2). The reachability condition of R_j is assumed in (3) and (4). (4) also implies that R_j's target evaluates to true. "false" and "error" do not apply to omitted rule target in (3) or omitted rule condition in (4). Consider the running example, the full decision coverage of the policy target requires a test to cover the following situations:
- *role=blue*
- $\neg$ *(role=blue)*, i.e., *role*$\neq$*blue*
- an error occurs when the match predicate for *role=blue* (lines 6-11) is evaluated (e.g., if the access request contains no value for attribute *role*)

The full decision coverage of the rule target of *max-drink-amount* requires one test to cover the following constraints:
- *role=blue* $\wedge \neg$ *(resource-id=Liquor* $\vee$ *resource-id=Medicine)* $\wedge$ *resource-id=Drink*
- *role=blue* $\wedge \neg$ *(resource-id=Liquor* $\vee$ *resource-id=Medicine)* $\wedge \neg$ *(resource-id=Drink)*
- *role=blue* $\wedge \neg$ *(resource-id=Liquor* $\vee$ *resource-id=Medicine)* $\wedge$ *an error occurs when resource-id=Drink is evaluated*

The full decision coverage of the rule condition of *max-drink-amount* requires one test to cover the following constraints:
- *role=blue* $\wedge \neg$ *(resource-id=Liquor* $\vee$ *resource-id=Medicine)* $\wedge$ *resource-id=Drink* $\wedge$ *amount*$\geq$*10*
- *role=blue* $\wedge \neg$ *(resource-id=Liquor* $\vee$ *resource-id=Medicine)* $\wedge$ *resource-id=Drink* $\wedge \neg$ *(amount*$\geq$*10)*
- *role=blue* $\wedge \neg$ *(resource-id=Liquor* $\vee$ *resource-id=Medicine)* $\wedge$ *resource-id=Drink* $\wedge$ *an error occurs* when *amount*$\geq$*10* is evaluated (e.g., if the access request contains no value for attribute *amount*)

The full decision coverage of rule *deny-liquor-medicine*' target, *resource-id=Liquor* ∨ *resource-id=Medicine*, requires one test to meet each of the following constraints in addition to the rule's reachability condition *role=blue*:

- *resource-id=Liquor* ∨ *resource-id= Medicine*
- ¬*(resource-id=Liquor* ∨ *resource-id= Medicine)*
- an error occurs when rule *max-drink-amount* is evaluated (e.g., if the access request contains no value for attribute *resource-id*).

Definition 2 has considered the occurrence of errors when a policy (set) target, a rule target, or a rule condition is evaluated. A variation of decision coverage is ***non-error decision coverage (NE-DC)***, where error tests are not considered.

3.3 Modified-Condition/Decision Coverage

MC/DC originated from NASA's RTCA/DO-178B document [16], which is "the primary means used by aviation software developers to obtain Federal Aviation Administration (FAA) approval of airborne computer software" [17]. DO-178B requires level-A software to achieve MC/DC of the software structure. Here a condition is a primitive Boolean valued expression that cannot be broken down into simpler Boolean expressions, whereas a decision is a Boolean-valued expression made up of conditions and logic operators (e.g., ∧, ∨, and ¬). Consider rule *deny-liquor-medicine*'s target as an example: *resource-id=Liquor* ∨ *resource-id= Medicine* is a decision. It is composed of two conditions *resource-id=Liquor* and *resource-id= Medicine*, and the logic operator ∨. Note that, here the term "condition" is different from "condition" in XACML rules. In addition to condition coverage (i.e., make a decision true and false at least once), MC/DC requires that: (1) every condition in a decision has taken on all possible outcomes at least once, and (2) each condition has been shown to independently affect the decision's outcome. For example, MC/DC of a conjunctive decision with n conditions (e.g., $c_1 \wedge ... \wedge c_n$) requires $n+1$ tests: one test that evaluates all conditions to true and n tests that evaluate one condition to false and other conditions evaluate to true. MC/DC of a disjunctive decision with n conditions (e.g., $c_1 \vee ... \vee c_n$) requires $n+1$ tests: one test that evaluates all conditions to false and n tests that evaluate one condition to true and other conditions evaluate to false.

In this paper, we apply MC/DC to each decision expression (i.e., policy set target, policy target, rule target, and rule condition) in XACML policies. We not only consider two truth values (i.e., true and false), but also error conditions.

Definition 3. A test suite *TS* for a policy set *PS* is said to satisfy *MC/DC* of *PS* if *TS* satisfies MC/DC of each decision expression:

(1) *TS* satisfies MC/DC of policy set target *pst*, and has a test to make *pst* evaluate to error,
(2) For each policy *Pi* in *PS*, *TS* achieves MC/DC of policy P_i's target pt_i, and has a test to make pt_i evaluate to error,
(3) For each rule R_j in each policy P_i of *PS*, *TS* achieves MC/DC of R_j's target and has a test to make R_j's target evaluate to error.
(4) For each rule R_j in each P_i of *PS*, *TS* achieves MC/DC of R_j's condition and has a test to make R_j's condition evaluate to error.

The reachability condition of P_i is assumed in (2), and the reachability condition of R_j is assumed in (3) and (4). (4) also implies that rule R_j's target evaluates to true. A variation of the above MC/DC is *non-error MC/DC (NE-MC/DC)*, where error tests are not considered.

Consider rule *deny-liquor-medicine*'s target: *resource-id=Liquor* ∨ *resource-id= Medicine*. Its MC/DC requires one test to meet each of the following constraints in addition to the rule's reachability condition *role=blue*:

- *resource-id=Liquor* ∧ ¬*(resource-id= Medicine)*, i.e., *resource-id=Liquor*
- ¬*(resource-id=Liquor)* ∧ *resource-id= Medicine*, i.e., *resource-id= Medicine*
- ¬*(resource-id=Liquor)* ∧ ¬*(resource-id= Medicine)*, i.e., *resource-id*≠*Liquor* ∧ *resource-id*≠ *Medicine*
- an error occurs when *resource-id=Liquor* ∨ *resource-id= Medicine* is evaluated, e.g., if the access request contains no value for attribute *resource-id*.

In this example, MC/DC requires one more test than decision coverage. Both of the first two tests make the expression true. Only one is needed to achieve the decision coverage.

3.4 Rule Pair Coverage (PC)

Policy combining algorithms and rule combining algorithms are meant to combining multiple conflicting decisions into a single access decision. Such conflicting decisions typically arise from different rules. Thus, testing may target the circumstances under which multiple rules evaluate to their specified effects.

Definition 4. A test suite *TS* for a policy set *PS* is said to achieve rule *Pair Coverage (PC)* of PS if, for each pair of rules within each policy P_i (excluding default rules), *TS* has a test to make both rules evaluate to their specified effects if feasible.

Because covering a pair of default rule and non-default rule would be the same as covering the non-default rule, Definition 4 excludes pairing of default rules. When there are default rules, rule pair coverage focuses on pairs of non-default rules. Note that, it is not always feasible to make two rules evaluate to their specified effects. In fact, different rules may deal with mutually exclusive circumstances. Consider rules *deny-liquor-medicine* and *max-drink-amount* in the running example. No test can satisfy the targets of both rules: *resource-id=Liquor* ∨ *resource-id=Medicine* and *resource-id=Drink*.

A variation of rule pair coverage is *Permit/Deny Rule Pair Coverage (PD-PC)*, where each rule pair consists of a permit rule and a deny rule. Testing may target such heterogeneous rule pairs in that homogeneous rule pairs do not necessarily yield conflicting decisions.

Definition 5. A test suite *TS* for a policy set *PS* is said to achieve *Permit/Deny rule Pair Coverage (PD-PC)* of PS if, for each pair of permit and deny rules within each policy P_i (excluding default rules), *TS* has a test to make both rules evaluate to their specified rule effects if feasible.

3.5 Relationships among Coverage Criteria

The aforementioned test coverage criteria are closely related to each other. Figure 2 shows the "subsumes" relationships between the coverage criteria. MC/DC subsumes decision coverage (DC) in that a test suite that achieves MC/DC always achieves the decision coverage. Similarly, decision coverage (DC) subsumes rule coverage, and rule pair coverage (PC) subsumes permit/deny rule pair coverage (PD-PC). In addition, MC/DC subsumes non-error (NE-) MC/DC, whereas decision coverage (DC) subsumes non-error decision coverage (NE-DC) because of the error tests. Error tests are syntactically valid access requests that make decision expressions (policy set target, policy target, and rule target/condition) evaluate to an *Indeterminate* decision. Since such an intermediate decision affects the next level decision of the containing policy element (policy set, policy, and rule), error tests are useful for detecting defects in XACML policies. Consider the following faulty policy with two deny rules (*Rule1* and *Rule2*) and a default permit rule (*Rule3*).

<RuleCombiningAlgId = Deny-overrides >
Rule 1: < name = "Tom", gender = "Male", deny>
Rule 2: < name = "Lee", class = "CS221", deny>
Rule 3: < , , deny>

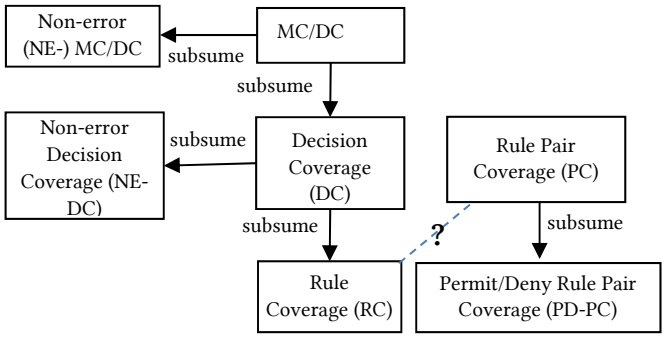

Figure 2: Relationships among test coverage criteria.

Suppose the given rule combing algorithm *Deny-overrides* is incorrect and the correct one is *Permit-unless-deny*. A non-error test that makes the two deny rules evaluate to either *NotApplicable* or *deny* requires that the access request contain a valid value for each of the three attributes: *name*, *gender*, and *class*. A request that contains no valid value for one of the attributes is an error test that would lead to an error occurrence. For example *{name = "Lee", gender="Male"}* is an error test because it contains no value for attribute *class*. *Rule 1* evaluates to *NotApplicable*, whereas *Rule 2* evaluates to *IndeterminateD*. The decision of the faulty policy is *IndeterminateD*. As the correct combining algorithm is *Permit-unless-deny*, however, the correct decision is permit. Thus, the above error test can reveal the fault. A non-error test makes *Rule 1* and *Rule 2* evaluate to either *NotApplicable* or deny. If either *Rule 1* or *Rule 2* evaluate to deny, the faulty and correct policies result in the same decision of deny. If both *Rule 1* and *Rule 2* evaluate to *NotApplicable*, the faulty and correct policies also yield the same decision of permit. Thus, non-error tests cannot reveal the fault. It is worth pointing out that the tests in this paper (including the error tests) target faults in the policy under test, rather than error in the XACML implementations as studied by [18]. They can be useful for testing XACML implementations, though. None of the existing research on testing XACML policies has discussed the concept of error tests. Note that rule pair coverage (PC) does not necessarily subsume rule coverage (RC) although each test that covers a pair of rules also covers the individual rules in the pair. As discussed before, tests for a specific rule pair may not exist.

3.6 Application of Coverage Criteria

The test coverage criteria can be applied as follows:

- Measuring test coverage adequacy of given tests. These tests may be produced by other testing methods when a policy is developed or represent actual access requests in a running system. As will be shown in Section 5, test suites of different coverage criteria have different levels of fault detection capabilities. Measurement of coverage adequacy of tests provides important guidelines for the development of access control tests. For instance, if MC/DC is required of a policy but the current tests are not yet adequate for MC/DC, then more tests need to be developed and executed. As test suites for rule coverage have a poor record in finding defects (refer to Section 5), a test suite that does not even achieve the rule coverage cannot assure the policy quality.
- Generating access requests automatically to meet a certain coverage criterion. The proposed coverage criteria provide guidelines for automated test generation when validating XACML policies under development. As described in Section 4, we have developed methods for generating tests to satisfy the coverage criteria. They make it possible to empirically evaluate the levels of quality assurance implies by the coverage criteria. In practice, when the proposed test generation methods are applied to an XACML policy, the tester needs to define the oracle value (i.e., expected response) of each test input in order to determine if the test passes or fails.

Because it is easy to implement the measurement of coverage criteria, this paper focuses on automated test generation for the coverage criteria and empirical evaluation of their effectiveness.

4 COVERAGE-BASED TEST GENERATION

The coverage-based test generators produce access requests from a given policy set or policy to satisfy a coverage criterion. They first collect the constraints on attributes according to the criterion as described in Section 3. Then they feed each constraint to the Z3-str constraint solver, and convert the result into an access request if the constraint is solved. Z3-str [19] is an extension to Z3 [20], an efficient SMT (Satisfiability Modulo Theories) Solver freely available from Microsoft Research. SMT generalizes boolean satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. Z3-str treats strings as a primitive type with common string operations.

The test generator for rule coverage aims to generate a test to cover each rule. It first composes the constraint for each rule in each policy of the given policy set and then uses Z3-str to generate an access request for the constraint. The algorithms are given below. The constraint for covering a rule includes policy set target, policy reachability condition, policy target, rule reachability condition, rule target and rule condition. Obtaining the reachability condition of each policy within the policy set and obtaining the reachability condition of each rule within a policy are described separately in Algorithms 2 and 3. Policy reachability depends on the policy combining algorithm in the policy set and rule reachability depends on the rule combining algorithm in each policy. They are reflected by the conditional statements in Algorithms 2 and 3, respectively.

Algorithm 1. *generateTestsForRuleCoverage(PS)*
Function: Generate tests for rule coverage
Input: Policy set $PS = < pst, pca, [P_1, P_2,..., P_m]>$
Output: A set of access requests Q

```
1   Q ← ∅
2   for each policy P_i = <pt_i, rca_i, [R_1, R_2,..., R_n]> in PS, do
3     constraint ← pst
4     constraint ← constraint ∧ policyReachability(PS, P_i)
5     constraint ← constraint ∧ pt_i
6     for each rule R_j =<rt_j, rc_j, re_j> in P_i, do
7         ruleConstraint ← constraint ∧ ruleReachability(P_i,R_j)
8         ruleConstraint ← ruleConstraint ∧ (rt_j ∧ rc_j)
9         Q ← Q∪{Z3-request(ruleConstraint)}
```

Algorithm 2. *policyReachability(PS, P_i)*
Function: Generate policy reachability constraint
Input: Policy set $PS = < pst, pca, [P_1, P_2,..., P_m]>$
Policy $P_i = <pt_i, rca_i, [R_1, R_2,..., R_n]>$
Output: *constraint*

```
1   constraint ← ""
2    for k=1 to i-1, do
3      if P_k's target pt_k is not empty
4          constraint ← constraint ∧ ¬ (pt_k)
5      else
6          if pca = Deny-overrides or Permit-unless-deny
7              rules ← all deny rules in P_k
8          else if pca = Permit-overrides or Deny-unless-permit
9              rules ← all permit rules in P_k
10         else if pca = First-applicable
11             rules ← all rules in P_k
12         for each rule R_s =<rt_s, rc_s, re_s> in rules, do
13             constraint ← constraint ∧ ¬ (rt_s ∧ rc_s)
```

Algorithm 3. *ruleReachability(P_i, R_j)*
Function: Generate rule reachability constraint
Input: Policy $P_i = <pt_i, rca_i, [R_1, R_2,..., R_n]>$
Rule $R_j =<rt_j, rc_j, re_j>$ in P_i
Output: *constraint*

```
1    constraint ← ""
2    if rca_i = Deny-overrides or Permit-unless-deny
3        rules ← all deny rules before R_j in P_i (s<j)
4    else if rca_i=Permit-overrides or Deny-unless-permit
5        rules ← all permit rules before R_j in P_i (s<j)
6    else if rca_i = First-applicable
7        rules ← all rules before R_j in P_i (s<j)
8    for each rule R_s =<rt_s, rc_s, re_s> in rules
9        constraint ← constraint ∧ ¬ (rt_s ∧ rc_s)
```

To deal with MC/DC and decision coverage, we use a truth table to manage its coverage status for each decision expression (e.g., policy set target, policy target, rule target, and rule condition). Each entry consists of three parts: a truth value for each of the basic conditions that comprise the decision expression, truth value of the decision expression (TRUE, FALSE, or ERROR), and whether the entry is covered by some existing test (TRUE or FALSE). Table 1 shows a sample MC/DC truth table for decision expression *resource-id=Liquor* ∨ *resource-id= Medicine*. The four entries are corresponding the four expected MC/DC tests discussed in Section 3.3. Entry 0 represents that *resource-id=Liquor* is True and *resource-id= Medicine* is False. In this case, the decision expression *resource-id=Liquor* ∨ *resource-id= Medicine* evaluates to True. "Covered" is False because no existing test has covered this entry. Entries 1-3 can represent tests for decision coverage. Let *conditions (constraint, i)* represent the conjunction of the basic conditions in the *i*-th entry. Consider entry 0 of Table 1. *conditions (resource-id=Liquor* ∨ *resource-id= Medicine, 0)* means *(resource-id=Liquor)* ∧ ¬*(resource-id=Medicine)*. Let *decision(constraint, i)* represent the decision in the *i*-th entry and *covered(constraint, i)* represent whether or not the *i*-th entry is covered by existing tests. For convenience, the truth table of an empty decision expression (e.g., policy set target, policy target, rule target, rule condition) has one entry, where *conditions(constraint, 0)* and *decision(constraint, 0)* are both *True*.

Table 1: MC/DC Truth Table for resource-id=Liquor ∨ resource-id= Medicine

Basic Conditions		Decision	Covered
resource-id =Liquor	resource-id = Medicine	resource-id=Liquor ∨ resource-id= Medicine	
TRUE	FALSE	TRUE	FALSE
FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE
		ERROR	FALSE

When dealing with a decision expression for test generation purposes, we check each entry of its MC/DC truth table. If an entry is already covered by the existing tests, no new test is needed for the entry. If it is not covered by any existing test and the truth value of the decision expression is TRUE, the conditions in this entry will be carried to the next decision expression (e.g., if the current expression is rule target, the next is rule condition). If it is not covered by any existing test and the truth value of the decision expression is FALSE or ERROR, a new test will be generated according to the entry's conditions together with all the constraints for reaching this decision expression.

Algorithm 4 below describes how to generate MC/DC tests for a policy set. Lines 2-7 initialize MC/DC truth tables. Then we deal with four levels of MC/DC truth tables for *pst* (policy set

target), pt_i (policy target of each policy), rt_j (rule target of each rule), and rc_j (for rule condition of each rule). pt_i is reachable only when *pst* is true (line 13). rt_j is reachable only when *pst* and pt_i are true (line 23). rc_j is reachable when if *pst*, pt_i, and rt_j are true (line 32). Therefore, at each level, we explore the next level only when the decision of the current entry is true. If the decision of current entry is false and it is not yet covered, we use Z3-str to generate a request: lines 9-10 for *pst*, lines 16-18 for pt_i, lines 26-28 for rt_j, line 35 for rc_j. If the decision of the current entry is error and is not yet covered, we generate an error request (lines 11-12 for *pst*, lines 19-21 for pt_i, lines 29-21 for rt_j, line 37 for rc_j). After a request is generated, we update the coverage information in all MC/DC truth tables. Note that generation of a normal or error request also involves policy reachability constraint (Algorithm 2) and/or rule reachability constraint (Algorithm 3).

Algorithm 4. *generateTestsForMC/DC(PS)*
Function: Generate tests for MC/DC
Input: Policy set $PS = < pst, pca, [P_1, P_2,..., P_m]>$
Output: A set of access requests Q
Steps:

```
1   Q ← ∅
2   create MC/DC truth table for pst
3   for each policy Pi = <pti, rcai, [R1, R2,..., Rn]> in PS, do
4     create MC/DC truth table for pti
5     for each rule Rj =( rtj, rcj, rej) in Pi do
6       create MC/DC truth table for rtj
7       create MC/DC truth table for rcj
8   for each entry u of pst's truth table
9     if decision(pst, u) = FALSE
10      Q ← Q∪{Z3-request(conditions(pst, u))}
11    else if decision(pst, u) = ERROR
2       Q ← Q∪{error-request(_, pst)}
13    else if decision(pst, u) = TRUE
14      for each Pi = <pti, rcai, [R1, R2,..., Rn]>, do
15        for each entry v of pti's truth table
16          if decision(pti, v)=FALSE & covered(pti, v)=FALSE
17              q ← Z3-request(conditions(pst, u) ∧ policyReachability(PS,Pi) ∧ conditions(pti, v))
18              add q to Q and update truth tables
19          else if decision(pti, v)=ERROR & covered(pti, v)=FALSE
20              q ←error-request(conditions(pst, u) ∧ policyReachability(PS,Pi), pti)
21              add q to Q and update truth tables
23          else if decision(pti, v) = TRUE
24            for each rule Rj =<rtj, rcj, rej>, do
25              for each entry w of rtj's truth table, do
26                if decision(rtj,w)=FALSE & covered(rtj,w)=FALSE
27                    q ← Z3-request(conditions(pst, u) ∧ policyReachability(PS,Pi)∧
                          ruleReachability(Pi, Rj)∧ conditions(pti, v)∧ conditions(rtj, w))
28                    add q to Q and update truth tables
29                if decision(rtj,w)=ERROR & covered(rtj,w)=FALSE
30                    q ←error-request(conditions(pst, u) ∧ policyReachability(PS,Pi) ∧
                          ruleReachability(Pi, Rj) ∧ conditions(pti, v), pti)
31                    add q to Q and update truth tables
32                else if decision(rtj,w)=TRUE
33                    for each entry z of rcj's truth table such that covered(rcj, z)=FALSE, do
34                      if decision(rcj,z)!=ERROR
35                          q←Z3-request(conditions(pst,u)∧ policyReachability(PS,Pi) ∧
                              ruleReachability(Pi, Rj) ∧ conditions(pti, v) ∧ conditions(rtj, w) ∧ conditions(rcj, z))
36                      else // decision(rcj, z) = ERROR
37                          q←error-request(conditions(pst,u)∧ policyReachability(PSi,P) ∧
                              ruleReachability(Pi, Rj)∧conditions(pti,v)∧ conditions(rtj,w), rcj)
38                      add q to Q and update tables
```

The test generation algorithm for decision coverage is a special case of Algorithm 4. Specifically, the MC/DC truth table in Algorithm 4 is replaced with the decision table. Consider *resource-id=Liquor* ∨ *resource-id= Medicine* whose MC/DC table is shown in Table 1. Its decision table has two columns as in Table 1: Decision and Covered. It has only one decision entry for TRUE, FALSE, and ERROR.

5 EMPIRICAL STUDIES

We have implemented our approach in a Balana-based tool. It has facilitated conducting empirical studies for evaluating scalability and cost-effectiveness of the coverage-based test generation methods. This section presents experiment setup and results and discusses threats to validity.

5.1 Experiment Setup

Our experiments use a total of seven cases studies (nine XACML3.0 policies) with different levels of complexity. As shown in Table 2, the number of rules ranges from a dozen to 1,280. Kmarket is the demonstration application of Balana with three individual policies and a total of 12 rules. All other policies are from the literature.

Table 2: Subject Policies

#	*Policy*	*#Rules*	*#Mutants (M14)*	*# Mutants (M8)*
1	Kmarket [15]	12	88	67
2	fedora[2]	12	94	58
3	conf [21]	15	107	64
4	itrust[3]	64	515	259
5	itrust5 [5]	320	2,563	1,283
6	itrust10 [5]	640	5,123	2,563
7	itrust20 [5]	1,280	10,243	5,123

Our approach to the evaluation of testing effectiveness is mutation analysis of subject policies. Mutation analysis is a widely applied technique for evaluating testing methods. The main hypotheses underlying mutation analysis [22] are: (a) the mutants are based on actual fault models and are representative of real faults, (b) developers produce programs (policies) that are close to being correct, (c) tests sufficient to detect simple faults (i.e., in mutants) are also capable of detecting complex faults. Recent experiments have confirmed that mutants are indeed similar to real faults for the purpose of evaluating testing techniques [23]. As discussed below, these hypothesis are valid for mutation analysis of XACML policies in this paper.

We generated mutants of each policy by using all the mutation operators in Table 3. Each mutant is a variant of the given policy with one fault injected by a mutation operator. The mutants generated by the mutation operators in Table 3 represent a great variety of faults in XACML policies. Each mutation operator may generate a number of mutants for a given policy. For example, given a policy with *n* rules, CRE (Change Rule Effect) creates *n* mutants because it creates a mutant by changing the effect of each rule. We use M14 and M8 to denote the set of all 14 mutation operators and the set of the first 8 mutation operators, respectively. M8 represents the set of mutation operators commonly used by the majority of the existing work on XACML policy testing. It is equivalent to the 11 mutation operators in [14] because the mutation operators in Table 3 are more coarse-grained. For instance, CRC (Change Rule/Policy Combining Algorithm. applies to both rule combining algorithm and policy combining algorithm. It represents two traditional mutation operators in [14]. M14 is based on the mutation operators in the XACML 2.0 mutant generator, XACMUT [24]. It does not include ANR (Add New Rule), RUF (RemoveUnique-nessFunction), AUF (AddUniqueness Function), CNOF (Change-N-OF-Function), and CLF (Change LogicalFunction). This paper evaluates the fault detection capabilities of the coverage-based test generation methods with both M14 and M8. Table 3 includes the numbers of non-equivalent mutants of each policy created by M14 and M8, respectively. It excludes those mutants that are equivalent to their original policy. As proven in [25], for example, the rule combining algorithms Permit-overrides and Deny-overrides make no difference with respect to a policy with permit-only (or deny-only) rules.

Table 3: Mutation Operators of XACML Policies

#	Fault Type	Mutation Operator: Name	Mutation Operator: Mutation
1	Incorrect policy	PTT*	set Policy/set Target True
2	/policy set target	PTF	set Policy/set Target False
3	Incorrect rule/ policy combining algorithm	CRC	Change Rule/Policy Combining algorithm
4	Incorrect rule effect	CRE	Change Rule Effect
5	Incorrect rule	RTT	set Rule Target True
6	target	RTF	set Rule Target False
7	Incorrect rule	RCT	set Rule Condition True
8	condition	RCF	set Rule Condition False
9		ANF	Add Not Function in condition
10		RNF	Remove Not Function in condition
11	Incorrect rule	FPR	First Permit Rules
12	ordering	FDR	First Deny Rules
13	Missing rule	RER	REmove a Rule
14	Missing target element	RPTE	Remove Parallel Target Element

Our experiments use the same protocol for each subject policy. First, we generate a test suite by each of the coverage-based test generation method. Second, we run each test suite against the given policy and record the actual response of each test. It will be used as the oracle value of this test when the policy mutants are tested. Third, we generate mutants of the policy using all the mutation operators in Table 3. Fourth, we run the test suite of each test generation method against each mutant. Since mutants represent the faults that likely occur in XACML policies, mutation score is considered the main indicator of the fault-detection capability.

5.2 Results

We present the experiment results from three perspectives: time performance of test generation, fault detection capability (i.e., effectiveness), and cost-effectiveness.

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

[3] http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=start

5.2.1 Time performance of test generation. Table 4 shows the number of tests generated by each test generation method for each subject policy. Typically, the test suite for decision coverage has more tests than that for rule coverage, whereas the MC/DC test suite may have many more tests than those of rule coverage and decision coverage. It depends on the complexity of policy/rule targets and rule conditions. For itrustX, pair coverage and permit/deny pair coverage are not applicable because the rules are all mutually exclusive.

Table 4: Number of Generated Tests

Subject	*RC*	*DC*	*NE-DC*	*MC/DC*	*NE-MC/DC*	*PC*	*PD-PC*
Kmarket	12	32	19	33	20	15	9
fedora	12	27	18	33	24	25	13
Conf	15	18	16	27	25	14	14
itrust	64	66	65	197	196	N/A	N/A
itrust5	320	322	321	983	982	N/A	N/A
itrust10	640	642	641	1,965	1,964	N/A	N/A
itrust20	1,280	1,282	1,281	3,929	3,928	N/A	N/A

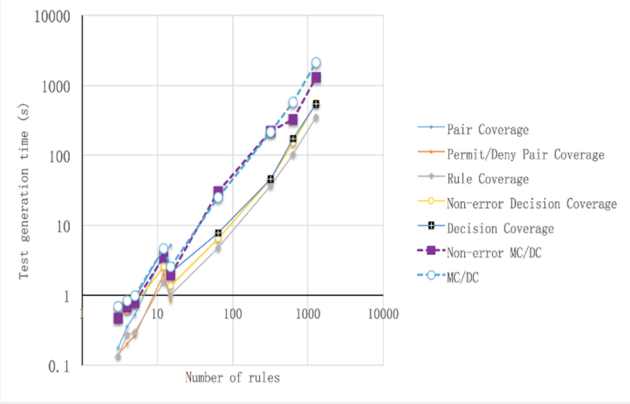

Figure 3: Test generation time.

Given a policy, the test generation time of a testing method is related to the total number of tests generated, which depends on the number and complexity of rules in the policy. Figure 3 shows the test generation time of each testing method with respect to the number of rules in all subject policies. The test generation was performed on a 64bit Ubuntu laptop (Inter Core i5-2410 @2.3 GHz, 3.8GB memory). Using a series of policies with similar rule structures (i.e., itrust, itrust5, itrust10, and itrust20) provides a good measurement of the scalability of test generation. Because MC/DC entails a much larger test suite than decision coverage and rule coverage, it also consumes more time for test generation. Nevertheless, the test generation time of each method is approximately linear to the number of rules. It is satisfactory even for the largest policy (i.e., itust20 with 1,280 rules). From the time performance perspective, all the test generation methods are applicable to large policies.

5.2.2 Fault detection capabilities. Table 5 shows the mutation scores of the coverage-based testing methods with respect to all subject policies. The mutation scores range from 50% to 63.6% for the rule coverage tests, from 62.5% to 96.6% for the decision coverage tests, and from 97.01% to 100% for the MC/DC tests, 58.2% to 70.3% for all rule pairs tests, and 43.1% to 70.3% for all PD-pairs tests. The results demonstrate that the rule coverage tests are far from adequate for high assurance of XACML policies because they cannot reveal many faults, and that the MC/DC tests are the most capable and provide high assurance of XACML policies. Further analysis of the mutant-killing results indicates that the rule coverage tests did not kill any of the mutants created by PTT, RTT, RCT or RPTE, and some of the mutants created CRC, FPR, and FDR. The decision coverage tests did not kill all of the mutants created by RTT, FPR, FDR, and RPTE. The MC/DC tests did not kill all of the mutants created RTT, FPR and FDR. Different from the rule coverage and decision coverage tests, they killed all RPTE mutants. The error tests for MC/DC and decision coverage killed some CRC and RPTE mutants.

Table 5: Mutation Scores (%) with M14

Subject	RC	DC	NE-DC	MC/DC	NE-MC/DC	PC	PD-PC
Kmarket	63.6	96.6	89.8	97.7	90.9	63.6	63.6
fedora	56.4	94.7	90.4	97.9	97.9	56.4	47.9
conf	55.1	86.9	72	100	98.1	55.1	55.1
itrust	49.9	62.9	62.7	100	100	N/A	N/A
itrust5	50	62.6	62.5	100	100	N/A	N/A
itrust10	50	62.5	62.5	100	100	N/A	N/A
itrust20	50	62.5	62.5	100	100	N/A	N/A
Average	50.1	63.0	62.9	99.98	99.94	57.9	54.9

Table 6: Mutation Scores (%) with M8

Subject	*RC*	*DC*	*NE-DC*	*MC/DC*	*NE-MC/DC*	*PC*	*PD-PC*
Kmarket	58.2	97.01	88.1	97.01	88.1	58.2	58.2
fedora	56.9	100	100	100	100	56.9	43.1
conf	70.3	100	96.9	100	96.9	70.3	70.3
itrust	74.5	100	100	100	100	N/A	N/A
itrust5	74.9	100	100	100	100	N/A	N/A
itrust10	75	100	100	100	100	N/A	N/A
itrust20	75	100	100	100	100	N/A	N/A
Average	**74.68**	**99.98**	**99.89**	**99.98**	**99.89**	**61.9**	**57.7**

Table 6 shows the mutation scores using M8. The mutation scores of the rule coverage tests are similar to those in Table 5. However, the decision coverage tests and the MC/DC tests have the same high mutation scores. The decision coverage tests are highly capable of killing M8 mutants, i.e., detecting the types of faults represented by all mutation operators in M8. Both the decision coverage tests and the MC/DC tests have higher mutation scores. For the three policies (i.e., fedora, conference, itrust) that are commonly used by the related work and this paper, the decision coverage tests and the MC/DC tests have killed all the mutants. They have outperformed the existing testing methods as described in the related work section.

Table 7: MKPT scores

Subject	*RC*	*DC*	*NE-DC*	*MC/DC*	*NE-MC/DC*	*PC*	*PD- PC*
Kmarket	4.67	2.66	4.16	2.61	4.00	3.73	6.22
Fedora	4.42	3.30	4.72	2.79	3.83	2.12	3.46
Conf	3.93	5.17	4.81	3.96	4.20	4.21	4.21
itrust	4.02	4.91	4.97	2.61	2.63	N/A	N/A
itrust5	4.00	4.98	4.99	2.61	2.61	N/A	N/A
itrust10	4.00	4.99	5.00	2.61	2.61	N/A	N/A
itrust20	4.00	5.00	5.00	2.61	2.61	N/A	N/A
Average	**4.01**	**4.94**	**4.99**	**2.61**	**2.62**	**3.11**	**4.44**

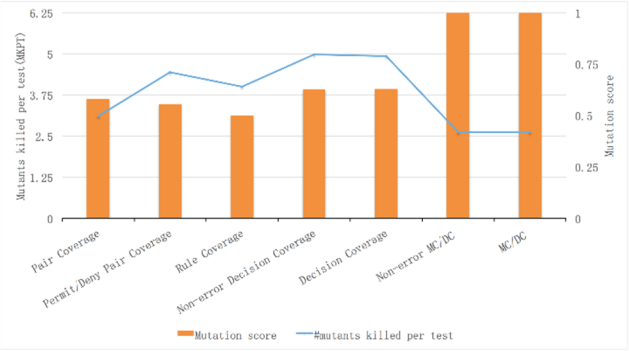

Figure 4: Cost-effectiveness of testing methods.

5.2.3 Cost-effectiveness. While mutation score is a good indicator of the fault detection capability of a testing method, it does not account for the testing cost. Ideally, we expect to find all potential faults in a given policy so as to achieve high assurance. In practice, this may be infeasible due to limited resources available (e.g., time and budget). Thus we need to take testing cost into consideration. Here we use the total number of tests created by a testing method as the main cost factor of testing because it often reflects the total test generation time and test execution time. We consider the average number of Mutants Killed Per Test (MKPT) as the indicator of cost-effectiveness. Table 7 shows MKPT scores for all coverage-based test generation methods. Figure 4 compares MKPT scores with mutation scores. While MC/DC is the most capable testing method in terms of mutation scores, it is not the most cost-effective. Although rule coverage is the least capable, it is more cost-effective than MC/DC. Among all the testing methods, decision coverage is the most cost-effective. The analysis of cost-effectiveness leads to the following observations: (1) when testing resources (e.g., time and budget) are very limited in the evolving process of policy development and validation, decision coverage (or even rule coverage) is a better choice than MC/DC. The tests for decision coverage (or rule coverage) should be performed first. (2) As the testing process progresses, it gets more and more expensive to find additional faults because more and more tests need to be created and executed. Nevertheless, before an XACML policy is deployed, a MC/DC-like test suite is high desirable to ensure correct enforcement of access control.

6 RELATED WORK

Martin et al. [13] defined several coverage measurements for XACML1.0/2.0 policies, such as policy hit percentage, rule hit percentage, and condition hit percentage. They used coverage information to reduce test suites produced by existing test generation methods. In comparison, this paper presents test criteria and methods for generating tests to satisfy the criteria. Decision coverage and MC/DC consider error tests for targets and conditions.

Cirg [11] generates access requests from counterexamples produced by Margrave [21] through the change-impact analysis of two synthesized versions. The difference of the two versions of a policy targets a test coverage goal, such as rule coverage or condition coverage. Because request generation from change-impact analysis may result in a large number of requests, Cirg reduces the number of tests by selecting tests based on policy structural coverage. The mutation scores of the testing methods in Cirg ranged from 30% to 60% in the case studies (except 100% for a trivial policy). Targen [12] derives access requests to satisfy all the possible combinations of truth values of the attribute id-value pairs found in a given policy. The mutation scores of Targen ranged from 75% to 79% for different case studies [7].

Access requests generated by Cirg and Targen typically use a limited number of subject, resource, action, and environment attributes. Generally, a request could use any combination of attributes. Because XACML requests are encoded in XML, they must be conforming to a specific XML Schema called the Context Schema. Bertolino et al., have developed the X-CREATE framework with multiple test generation algorithms by considering the structures of the Context Schema [9]. These algorithms can generate requests that use more than one subject, resource, action, or environment attribute. The mutation scores of the testing methods in X-CREATE ranged from 75% to 96% for several small policies. Bertolino et al., have also developed other test selection strategies, such as Simple Combinatorial and Incremental XPT [7]. The mutation scores ranged from 3% to 100%, whereas the mutation scores of the Incremental XPT strategy ranged from 55% to 100%. Bertolino et al., [8] proposed an approach to select tests from a given large test suite based on the rule coverage criterion. It selects tests to match each rule target set, which is the union of the target of the rule and all enclosing policy and policy sets targets. The mutation scores of this approach ranged from 62% to 98%. Our paper proposes several additional criteria. Our empirical studies show that decision coverage and MC/DC are more capable. Li et al., [10] have developed XPTester, which used a symbolic execution technique to generate requests from XACML policies. They convert the policy under test into semantically equivalent C Code Representation (CCR) and symbolically execute CCR to create test inputs and translate the test inputs to access requests. This approach has the same fault detection capability as the Preliminary XPT algorithm in the X-CREATE framework, but it produces smaller test suites. The mutation scores of XPTester ranged from 37% to 93% in the case studies.

This paper is different from the above work on test generation from XACML policies [6]-[14]. First, this paper targets XACML 3.0, whereas the above work all deals with earlier versions of XACML (i.e., 2.0 or 1.0). Second, this paper aims at stronger coverage criteria, i.e., decision coverage and MC/DC, whereas the existing work is only concerned with rule coverage. As discussed before, a test suite for rule coverage can be very weak at fault detection. In addition to mutation scores, this paper also provides an analysis of cost-effectiveness using average number of mutants killed per test.

Safarzadeh et al. have proposed a model-based approach for testing the implementation of access control in a system, where system functional model and access control policy are specified by extended finite state machines and XACML, respectively. This approach derives conditions from rules in the policy and the functionality and applies MC/DC to the conditions for test generation. Our approach does not rely on functional models.

7 CONCLUSIONS

We have described the test coverage criteria for XACML3.0 policies and efficient methods for generating tests to satisfy each of the coverage criteria. We have also presented the empirical studies for evaluating the scalability, fault detection capabilities, and cost-effectiveness of the coverage-based test generation methods. The results show that the coverage-based tests, especially MC/DC tests, can achieve high assurance of access control enforcement.

The empirical studies also indicate that some mutants may not be killed by a given coverage-based test suite. It is highly desirable to conduct a theoretical analysis on why they are not killed. One approach is to formalize the fault detection conditions of these mutants which must be satisfied by the tests in order to kill them. It is also interesting to investigate whether the coverage-based test suites can be reduced while maintaining the same level of fault detection capability. Our future work will also exploit the proposed coverage criteria to select tests from large test suites generated by other testing methods. For example, combinational test generation for XACML policies may produce a large number of tests. Not all of them make contributions to the fault detection. The coverage criteria can be used to select tests in order to improve cost-effectiveness.

ACKNOWLEDGMENTS

This work was partially supported by the US National Science Foundation under grants 1359590 and 1618229. The mutation tool was implemented by Mr. Jimmy Wang and Mr. Shuai Peng.

REFERENCES

[1] OASIS. 2013. *eXtensible Access Control Markup Language (XACML)* Version 3.0, (Jan. 2013), http://www.oasisopen.org/committees/xacml/

[2] N. Li, J.C. Mitchell, W.H. Winsborough. 2002. Design of a Role-Based Trust Management Framework. In *Proc. IEEE Symposium on Security and Privacy* (S&P 2002). 114–130.

[3] V. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone. 2013. *Design of a Role-Based Trust Management Framework.* NIST Special Publication 800-162. (October 2013).

[4] FEDCIO2: *Federal Identity, Credential, and Access Management (FICAM) Roadmap and Implementation Guidance Version 2.0.* (December 2011).

[5] D. Xu, N. Shen, and Y. Zhang. 2015. Fault-Based Testing of Combining Algorithms in XACML3.0 Policies. In *Proc. of the 27th International Conf. on Software Engineering and Knowledge Engineering* (SEKE'15), Pittsburg, (July 2015).

[6] A. Bertolino, F. Lonetti, and E. Marchetti. 2010. Systematic XACML Request Generation for Testing Purposes. In *EUROMICRO Conference on Software Engineering and Advanced Applications* (SEAA), Lille, (September 2010).

[7] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti. 2012. Automatic XACML Requests Generation for Policy Testing. In *The Third International Workshop on Security Testing* (SecTest 2012), Montreal, (April 2012).

[8] A. Bertolino, Y. Le Traon, F. Lonetti, E. Marchetti, and T. Mouelhi. 2014. Automatic XACML Requests Generation for Policy Testing. In *IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 12-21. (2014),

[9] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti. 2012. The X-CREATE Framework-A Comparison of XACML Policy Testing Strategies. In *Proc. of the 8th International Conference on Web Information Systems and Technologies* (WEBIST). 155–160.

[10] Y. Li, Y. Li, L. Wang, G. Chen. 2014. Automatic XACML Requests Generation for Testing Access Control Policies. In *Proc. of the 26th International Conf. on Software Engineering and Knowledge Engineering* (SEKE'14). Vancouver. (July 2014).

[11] E. Martin, and T. Xie. 2007. Automated Test Generation for Access Control Policies via Change-Impact Analysis. In *Proc. 3rd International Workshop on Software Engineering for Secure Systems* (SESS). 5-11.

[12] E. Martin, and T. Xie. 2006. Automated Test Generation for Access Control Policies. In *Supplemental Proc. of ISSRE.* (Nov. 2006).

[13] E. Martin, and T. Xie. 2006. Defining and Measuring Policy Coverage in Testing Access Control Policies. In *Proc. of the 8th International Conf. on Information and Communications Security.* 139-158, (December 2006).

[14] E. Martin, and T. Xie. 2007. A Fault Model and Mutation Testing of Access Control Policies. In *Proc. of the 16th International Conf. on World Wide Web* (WWW'07). 667-676, (May 2007).

[15] WSO2. 2012. *Balana: An Open Source XACML 3.0 Implementation.* http://xacmlinfo.org/2012/08/16/balana-the-open-source-xacml-3-0-implementation/

[16] RTCA/DO-178B. 1992. *Software Considerations in Airborne Systems and Equipment Certification.* RTCA, Inc., Washington, D. C., (December 1992).

[17] K. Hayhurst, D. Veerhusen, J. Chilenski, L. Rierson, 1992. *A Practical Tutorial on Modified Condition/Decision Coverage.* NASA. (May 2001).

[18] N. Li, J.H. Hwang, T. Xie. 2008. Multiple-Implementation Testing for XACML Implementations. In *Proc. of the Workshop on Testing, Analysis, and Verification of Web Services and Applications* (TAV-WEB'08), 27-33.

[19] Y. Zheng, X. Zhang, V. Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proc. of the 9th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE'13). 114-124.

[20] L. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. of the 14th International Conference Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'08), LNCS volume 4963. Springer (2008).

[21] K. Fisler, S. Krishnamurthi, L.A. Meyerovich, and M.C. Tschantz. 2005. Verification and Change-Impact Analysis of Access-Control Policies. In *Proc. of the 27th International Conference on Software Engineering* (ICSE'05). 196-205.

[22] Y. Jia, and M. Harman. 2010. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. on Software Engineering*, Vol. 37, No. 5, 649-678.

[23] R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, and G. Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing? In *Proc. of the Symposium on the Foundations of Software Engineering* (FSE'14), 654-665, Hong Kong, (November 2014).

[24] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, 2013. XACMUT: XACML 2.0 Mutants Generator. In *Proc. of 2013 IEEE Sixth International Conf. on Software Testing, Verification and Validation Workshops.* 28-33.

[25] D. Xu, Y. Zhang, N. Shen, 2015. Formalizing Semantic Differences between Combining Algorithms in XACML 3.0 Policies. In *Proc. of the 2015 International Conference on Software Quality, Reliability and Security* (QRS'15), Vancouver, Canada (August 2015).

[26] M. Safarzadeh, M. Taghizadeh, B. Zamani, and B. T. Ladani. 2017. An Automatic Test Case Generator for Evaluating Implementation of Access Control Policies. *The ISC International Journal of Information Security.*