Automated Strong Mutation Testing of XACML Policies

Dianxiang Xu
Department of Computer Science Electrical Engineering
University of Missouri Kansas City
Kansas City, MO 66410, USA
dxu@umkc.edu

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

ABSTRACT

While the existing methods for testing XACML policies have varying levels of effectiveness, none of them can reveal the majority of policy faults. The undisclosed faults may lead to unauthorized access and denial of service. This paper presents an approach to strong mutation testing of XACML policies that automatically generates tests from the mutants of a given policy. Such mutants represent the targeted faults that may appear in the policy. In this approach, we first compose the strong mutation constraints that capture the semantic difference between each mutant and its original policy. Then, we use a constraint solver to derive an access request (i.e., test). The test suite generated from all the mutants of a policy can achieve a perfect mutation score, thus uncover all hypothesized faults or demonstrate their absence. Based on the mutation-based approach, this paper further explores optimal test suite that achieves a perfect mutation score without duplicate tests. To evaluate the proposed approach, our experiments have included all the subject policies in the relevant literature and used a number of new policies. The results demonstrate that: (1) it is scalable to generate a mutation-based test suite to achieve a perfect mutation score, (2) it can be impractical to generate the optimal test suite due to the expensive removal of duplicate tests, (3) different from the results of the existing study, the modified-condition/decision coverage-based method, currently the most effective one, has low mutation scores for several policies.

CCS CONCEPTS

• **Security and privacy** → **Access control**; *Software security engineering*; Authorization.

KEYWORDS

access control; XACML; mutation testing; constraint solving; test generation

ACM Reference Format:
Dianxiang Xu and Roshan Shrestha, Ning Shen. 2020. Automated Strong Mutation Testing of XACML Policies. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies (SACMAT '20), June 10–12, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3381991.3395599

SACMA T '20, June 10–12, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7568-9/20/06…$15.00
https://doi.org/10.1145/3381991.3395599

1 INTRODUCTION

While high expressiveness of XACML allows for the specification of fine-grained attribute-based access control policies, its increased complexity has raised concerned about the likelihood of policy faults. Such faults may lead to unauthorized access and denial of service if a policy is not thoroughly validated [23]. As an important practice for quality assurance of access control systems, policy testing has recently gained much attention. The basic idea is to execute the policy under test against a test suite (i.e., a set of test cases) with the intent to find faults. Each test case consists of a test input (i.e., access request) and an oracle value (i.e., correct response). The policy is faulty if one of the tests fails, i.e., the policy's actual response to the request is different from the oracle value. Several methods have been proposed to automatically generate test inputs from XACML policies [1–5, 15–18, 23], where oracle values need to be defined manually according to the access control requirements. To evaluate the effectiveness of these methods, a common approach is mutation analysis. It creates a set of mutants of a correct policy and then exercises each mutant against the tests generated by the testing method under evaluation. A mutant is a modified version of the given policy where a policy element is mutated according to a fault model of representative types of errors. A mutant is said to be killed if at least one test fails, otherwise it is a live mutant. A live mutant is either faulty or functionally equivalent to the original policy (called equivalent mutant). The effectiveness of the testing method is measured by its mutation score, i.e., the ratio between the total number of mutants killed and the total number of non-equivalent mutants. Experimental studies have shown that the existing methods have varying levels of effectiveness in terms of their mutation scores (refer to Section 2.2).

The above research, however, has two issues. First, none of the existing methods can reveal the majority of faults seeded in XACML policies. Even for the most effective MC/DC method, our study in Section 7 shows that its mutation scores can be as low as 50%. If these methods are applied to a real-world system, they may leave numerous access control vulnerabilities in the deployed system. So, is it possible to achieve a 100% mutation score, at least for all the XACML policies studied so far? Second, to calculate the mutation score in a mutation analysis experiment, we need to examine each live mutant manually to determine if it is equivalent to the original policy. This can be a daunting task when there are a large number of live mutants. For example, the mutation scores of 50%-63.6% for the rule-coverage testing [23] indicate that 36.4%-50% of all the mutants in the experiments had to be inspected.

To address these issues, this paper presents an approach to strong mutation testing that automatically generates test inputs from the mutants of a given policy. For each mutant, the approach aims to

find an access request to which the mutant and its original policy will respond differently – if the given policy is correct, then the mutant is faulty, and vice versa. To do so, we first compose the strong mutation constraints that capture the semantic difference between the mutant and its original policy. They include: (a) reachability constraint that any request must satisfy in order to trigger the evaluation of the mutated policy element; (b) necessity constraint that any request must satisfy in order to make the mutated policy element and the original policy element evaluate to different immediate decisions; (c) propagation constraint that any request must satisfy in order to make the mutant and the original policy produce different policy-level decisions. Once these constraints are obtained and analyzed, we feed them to a constraint solver (e.g., Z3 [6]) to find an access request. As such, we can generate a set of tests from all mutants of the given policy. Note that, solving the above reachability, necessity, and propagation constraints of all XACML policies is in general a hard problem due to the complexity of XACML policies. This is similar to the existing methods that rely on a constraint solver for test generation from XACML policies[22, 23]. Nevertheless, our experiments in Section 7 have shown that the mutation-based test generation approach is scalable for all the policy examples. The test suite generated directly from all mutants of each policy can kill all non-equivalent mutants and thus achieve a 100% mutation score. *This has addressed the first issue.* A mutant is non-equivalent to the original policy if the mutation-based test generation approach returns an access request, otherwise it is an equivalent mutant in that the reachability, necessity, and propagation constraints are unsatisfiable. Here we assume that the constraint solver is sound and complete for all types of solvable constraints in each given policy. Therefore, the mutation-based test generation approach can automatically determine whether or not a mutant is equivalent to the original policy. *This has addressed the second issue.*

In practice, the test suite generated by the proposed mutation-based approach can most effectively assure the quality of the policy even though the policy is not yet known correct or faulty before it is tested. If the policy passes all the mutation-based tests, then it is free from all the faults represented by the mutants. If it fails a test, the failure not only indicates that the given policy is faulty, but also suggests a possible fix because the mutated policy element in the mutant from which the test was generated might be a correct version of the corresponding policy element.

This paper is the first work on mutation-based test generation for XACML policies. In comparison, the existing work has used policy mutants only for evaluating the effectiveness of testing methods in terms of mutation scores. The notions of reachability, necessity, and propagation constraints originate from program mutation testing or constraint-based software testing [7–9]. Because the problem with propagation constraint of software is known to be intractable due to the explosion of program execution paths and the halting problem [8, 9], the existing mutation-based software testing methods only deal with reachability and necessity constraints. This is also known as weak mutation testing [11], which cannot achieve 100% mutation score and cannot automatically determine whether live mutants are equivalent mutants [8, 25]. In fact, this paper is the first work on the implementation of strong mutation testing that integrates all reachability, necessity, and propagation constraints. It is feasible because of the unique features of XACML policies, especially without loop structure.

A further contribution of this paper is the investigation of optimal test suites of XACML policies. A test suite for a given policy is said to be optimal if it achieves a 100% mutation score without any duplicate tests – every test kills at least one unique mutant that is not killed by any other test in the test suite. Based on the mutation-based approach, we have developed an algorithm for generating such optimal test suites. This enables quantitative comparison of cost-effectiveness with the existing testing methods. Although mutation score is commonly used as the primary indicator of effectiveness (i.e., fault detection capability), a testing method with a higher mutation score is not necessarily more cost-effective. As demonstrated in [23], for example, the MC/DC method has much higher mutation scores than the rule coverage method. It is also more expensive because it produces many more tests. The MC/DC method is actually less cost-effective from the perspective of mutants killed per test (MKPT). This paper presents an empirical comparison between the proposed strong mutation approach and the existing methods in terms of MKPT score.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 introduces XACML policies and policy mutation. Section 4 describes the basic constraint-based test generation techniques in our approach. Section 5 presents test generation from individual policy mutants. Section 6 discusses mutation-based generation of optimal test suites. Section 7 presents the experiments. Section 8 concludes this paper.

2 RELATED WORK

2.1 Mutation Testing

Mutation testing, originated from the area of software testing, is a widely applied technique for the empirical evaluation of software testing methods [12]. Mutation analysis of a program under test is to mutate the program to various faulty versions, i.e., mutants. Each mutant has one fault injected by a mutation operator. Mutation operators are defined with respect to a fault model, which is a collection of fault types in the programming language. The main hypotheses of mutation testing [20] include: (a) the mutants are based on actual fault models and are representative of real faults, (b) developers produce programs that are close to being correct, and (c) tests sufficient to detect simple faults (i.e., in mutants) are also capable of detecting complex faults. Experiments have shown that mutants are indeed similar to real faults for the purpose of evaluating testing techniques [13].

The existing work on software mutation testing falls into three categories: (a) infrastructure and tool support such as fault models, mutation operators, and test execution and analysis of mutants, (b) empirical evaluation of the fault detection capability of a testing method in terms of mutation score. It typically applies the testing method to the mutants of given subject programs, and (c) mutation-based test generation, which derives a test from one or more mutant of a given program so that the mutant and its original program produce different execution results. Such a test needs to meet the reachability, necessity, and propagation constraints. The existing techniques primarily follow the concept of weak mutation testing [11] that uses the reachability and necessity constraints to generate

test inputs [25]. The main reason is that it is intractable to solve the propagation constraint [8].

This paper aims at strong mutation testing of XACML policies that deals with all reachability, necessity, and propagation constraints for automated test generation. Different from general-purpose programming languages, XACML has no loops and thus does not suffer from the explosion of execution paths or the halting problem. The propagation constraint primarily depends on the rule and policy combining algorithms. Although the various combining algorithms in XACML are complicated, it is tractable to deal with the propagation constraint as demonstrated in this paper. The existing work on mutation testing of XACML policies focuses on mutation tools and evaluation of testing methods with policy mutants. Similar to software mutation testing, a major challenge is to determine whether or not a live mutant is equivalent to the original policy. This is usually done by manual review and analysis of the mutant code. In comparison, this paper provides a way to automatically determine equivalent mutants: a mutant is equivalent to its original policy if and only if its reachability, necessity, and propagation constraints are not satisfiable. If it is not equivalent, our approach will result in a test that demonstrates the difference.

2.2 Testing of XACML Policies

The existing approaches to test generation for XACML policies fall into two categories: model-based testing that derives tests from models (i.e., black-box testing), and policy-based testing that produces test inputs directly from the policy under test (i.e., white-box testing). As access control policies are extra constraints on system functions, the model-based testing approach usually integrates functional models with access control specifications, and can generate both test inputs and oracle values. Safarzadeh et al. [21] have proposed to specify system functions and access control policies by extended finite state machines and XACML, respectively. They derive test conditions from the state machines and the rules in the XACML policy and then apply MC/DC to the conditions for test generation. Khamaiseh et al. [14] proposed a model-based testing method for obligatory ABAC systems, where access control policies are implemented in XACML. Given a test model that consists of a functional Petri net model and an obligatory ABAC policy, the approach weaves the policy with the functional model into an integrated model so as to generate access control tests. Although the above methods have both involved XACML, the system implementation tested by them may or may not rely on XACML. How to build effective test models, however, remains a critical challenge.

This paper is primarily related to the work that generates test inputs from the XACML policy under test. Compared to the model-based testing, this approach does not produce the oracle value of each test. Mutation analysis of XACML policies had been commonly used to evaluate testing effectiveness. Cirg [17] generates test inputs from the counterexamples produced by a model checker 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. Mutation scores range from 30% to 60% in different case studies (except 100% for a simple policy). Targen [16] derives test inputs to satisfy all the possible combinations of truth-values of the attribute id-value pairs found in a given policy. The mutation scores range from 75% to 79% for different policies [4]. Considering that requests must conform to the XML Context Schema, Bertolino et al. have developed the X-CREATE framework for dealing with the structures of the Context Schema [2]. Mutation scores range from 75% to 96% for several small policies. They have also developed other test selection strategies, such as Simple Combinatorial and Incremental XPT [1]. Mutation scores of Simple Combinatorial range from 3% to 100%, whereas mutation scores of Incremental XPT ranged from 55% to 100%. Bertolino et al. [5] proposed an approach to selecting tests based on the rule coverage criterion. It chooses existing tests to match each rule target set, which is the union of the target of the rule and all enclosing policy and policy set targets. Mutation scores of this approach range from 62% to 98%. XPTester uses symbolic execution technique to generate test inputs from XACML policies [15]. It converts the policy under test into a semantically equivalent C Code Representation (CCR) and symbolically executes CCR to create test inputs. Mutation scores of XPTester range from 37% to 93%. The above methods are all based on earlier versions of XACML (1.0 or 2.0).

XPA (XACML Policy Analyzer) is an integrated toolkit for testing, mutating, and debugging XACML 3.0 policies [22–24]. XPA offers a number of coverage-based test generators[23]. Mutation scores of the rule coverage and decision coverage methods range from 50% to 63.6%, and from 62.5% to 96.6, respectively. The MC/DC method has achieved a mutation score of 97.7% or higher although it did not kill all mutants for all subject policies. The fault-based testing of combining algorithms aims to generate tests for revealing incorrect uses of combining algorithms [22]. This approach falls into mutation-based test generation, but is limited to only one mutation operator.

3 MUTATION OF XACML POLICIES

3.1 XACML Policies

This paper is based on XACML 3.0, the current version of the standard [19]. For convenience, this paper focuses on policies, rather than policy sets, although our implementation has covered both. A policy P is a triple $< PT, RCA, R >$, where PT is the policy target, RCA is the rule-combining algorithm, and R is the list of rules. Each rule $r \in R$ is a triple $< rt, rc, re >$, where rt is the rule target, rc is the rule condition, and $re \in \{Permit, Deny\}$ is the rule effect. $< rt, rc, Permit >$ is called a permit rule, whereas $< rt, rc, Deny >$ is a deny rule. The target of a policy (or rule) specifies the set of requests to which the policy (or rule) is intended to apply. It is represented as 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. The condition of a rule refines the applicability of the rule established by the rule target.

We use the following *Sample-PO* policy as a running example. For readability purposes, it is presented in plain text without attribute types and categories. $Allof$ and $AnyOf$ clauses are replaced with traditional logical operators "*and*" and "*or*". The attributes are

department, *title*, *location*, and *job-class*. The policy target is defined over the *department* attribute, whereas the four permit rules and two deny rule are defined over *title*, *location* and *job-class*.

Policy name: *Sample-PO*
Policy target: *department = "HR" or department = "IT"*
Rule-Combining Algorithm: *Permit-overrides*
Rules:
R1: *<title="director", location="on-campus", Permit>*
R2: *<title="director", location="off-campus", Permit>*
R3: *<title="deputy-director", location="on-campus", Permit>*
R4: *<title="deputy-director", location="off-campus", Permit>*
R5: *<job-class="guest" or job-class="part-time", location="off-campus", Deny>*
R6: *<job-class="intern" or job-class="contractor", location="off-campus", Deny>*

An access request consists of attribute names, categories, values, and types. Unless explicitly specified, we use a set of attribute name and value pairs to represent a request, assuming that the attribute categories and types are correct. For example, {*department = "HR", title="deputy-director", location="on-campus"*} is a valid request of the running example. Note that a valid request may cause the occurrence of a runtime error for different reasons, such as mismatch of an attribute type and an exception of expression and function evaluation. Consider {*department = "HR"*}. If the category of attribute *department* and the type of value *"HR"* match those in the running policy, the policy target will evaluate to true, otherwise its evaluation leads to an error occurrence. This is similar for rule target and rule condition. Error handling has intricate implications on the semantics of policies and the evaluation of access decisions.

Given a logical expression ω (policy target, rule target, or rule condition) and a request q, we use ω, $\neg\omega$, and $error(\omega)$ to represent that ω evaluates to true (i.e., there is a match if ω is a target), false (i.e., no-match if ω is a target), and indeterminate (i.e., error occurrence), respectively. Given rule $r =< rt, rc, re >$, let $fire(rt, rc)$ denote $rt \wedge rc$, $notApplicable(rt, rc)$ denote $\neg rt \vee (rt \wedge \neg rc)$, and $error(rt, rc)$ denote $error(rt) \vee (rt \wedge error(rc))$, and $notFire(rt, rc)$ denote $notApplicable\,(rt, rc) \vee error(rt, rc)$. According to the XACML 3.0 specification [19], rule-level and policy-level decisions are formalized by Definitions 1 and 2, respectively.

Definition 1. Given a rule $r =< rt, rc, re >$ and an access request q, the rule decision, denoted as $d(r, q)$, is defined as:

$$\begin{cases} Permit & \text{if } re = Permit \wedge fire(rt, rc), \\ Deny & \text{if } re = Deny \wedge fire(rt, rc), \\ N/A & \text{if } notApplicable(rt, rc), \\ ID & \text{if } re = Deny \wedge error(rt, rc), \\ IP & \text{if } re = Permit \wedge error(rt, rc). \end{cases}$$

where N/A, ID, and IP denote *Not-applicable*, *Indeterminate Deny*, and *Indeterminate Permit*, respectively.

Definition 2. Given policy $P =< PT, RCA, R >$ and an access request q, the policy decision, denoted as $d(P, q)$, is defined as:

$$\begin{cases} N/A & \text{if } \neg PT, \\ rca(RCA, R, q) & \text{if } PT, \\ rca(RCA, R, q) & \text{if } error(PT) \wedge rca(RCA, R, q) \in \{N/A, ID, IP, IDP\}, \\ IP & \text{if } error(PT) \wedge rca(RCA, R, q) = Permit, \\ ID & \text{if } error(PT) \wedge rca(RCA, R, q) = Deny. \end{cases}$$

Table 1: Mutation Operators

No	Name	Meaning	Fault Type
1	PTT	set Policy Target True	Incorrect policy target
2	PTF	set Policy Target False	
3	CRC	Change RCA	Incorrect RCA
4	CRE	Change Rule Effect	Incorrect rule effect
5	RTT	set Rule Target True	Incorrect rule target
6	RTF	set Rule Target False	
7	RCT	set Rule Condition True	Incorrect rule condition
8	RCF	set Rule Condition False	
9	ANF	Add Not in condition	
10	RNF	Remove Not in condition	
11	FPR	First Permit Rule	Incorrect rule ordering
12	FDR	First Deny Rule	
13	RER	REmove a Rule	Missing rule
14	RPTE	Remove Parallel Element	Missing target element

where $rca(RCA, R, q)$ represents the combined decision of all rules in R with respect to q by the rule-combining algorithm *RCA*, and *IDP* denotes *Indeterminate Deny/Permit*.

The main rule combining algorithms in XACML 3.0 include *Permit-overrides*, *Deny-overrides*, *Permit-unless-deny*, *Deny-unless-permit*, *First-applicable*, *Ordered-permit-overrides*, and *Ordered-deny-overrides*. Their semantics are explained in the standard specification and can be formulated by decision tables [10]. This paper will elaborate on five of them because *Ordered-permit-overrides* is similar to *Permit-overrides* and *Ordered-deny-overrides* is similar to *Deny-overrides*.

3.2 Mutation Operators

This paper follows the fault model and mutation operators of XACML 3.0 in the recent literature [23], as shown in Table 1. The fault types include incorrect policy (policy set) target, incorrect rule-combining (policy-combining) algorithm, incorrect rule effect, incorrect rule target, incorrect rule conditions, incorrect rule ordering, missing rule, and missing a parallel target element (i.e., $AnyOf$ or $AllOf$ clause). Some of the fault types are named based on the assumption that the policy under mutation is correct. For example, RPTE ("Remove Parallel Target Element") creates mutants with a fault of missing target element by removing a parallel $AnyOf$ or $AllOf$ clause in a correct rule (policy or policy set) target. In this paper, however, the mutation-based test generation approach aims at finding access request q such that policy P and its mutant P' produce distinct responses, i.e., $d(P, q) \neq d(P', q)$. It covers the case where P is incorrect and P' is correct. In this case, the fault type in P is not directly named from the mutation operator that has obtained P' from P. Instead, it should be reflected by the opposite mutation operator that mutates P' to P. Generally if P' is a mutant of P, then P is also a mutant of P'. The underlying mutation operator is the opposite. Consider RER ("REmove a Rule") and RPTE in Table 1, their opposite mutation operators, "Add a Rule" and "Add Parallel Target Element", are undefined. This paper is applicable to both directions of mutation. It covers the faults that can be created by the reversed version of each operator, if unspecified, in Table 1.

Table 2: Sample Mutants of the *Sample-PO* Policy

Name	Mutated Element
Sample-PO-PTT	Policy target: true
Sample-PO-RPTE0-1	Policy target: department = "HR"
Sample-PO-CRE3	R3:<title="deputy-director", location="on-campus", Deny>
Sample-PO-RTT5	R5: <true, location="off-campus", Permit>
Sample-PO-RPTE5-1	R5: <job-class="guest", location="off-campus", Deny>
Sample-PO-RCT3	R3: <title="deputy-director", true, Permit>
Sample-PO-CRC-FA	Rule-combining algorithm: First-applicable

Each mutant is named after the original policy, the mutation operator, and the indices of the mutated element if applicable. For a RPTE mutant, the first index is to specify whether it is applied to the policy target (0) or a rule and the second index refers to the removed parallel target element. Table 2 shows several mutants of the running example. *Sample-PO-PTT* is obtained by changing the policy target to true (i.e., the policy target is removed). *Sample-PO-RPTE0-1* is created by applying RPTE to the policy target's second parallel element. *Sample-PO-CRE3* results from applying CRE to rule 3, which changes the effect from *Permit* to *Deny*.

4 CONSTRAINT-BASED GENERATION OF POLICY TESTS

In this section, we introduce the basic idea of constraint-based test generation in the mutation-based approach.

4.1 Test Generation with Non-Error and Error Constraints

Let $normalTest(\phi)$ represent an access request that satisfies a non-error constraint ϕ, which does not involve error occurrence. For example, *(department="HR"* $\vee$ *department ="IT")* $\wedge$ *title="deputy-director"* $\wedge$ *location="on-campus"* is a non-error constraint. Using a constraint solver, *normalTest* would return an access request like the following: {*department="HR", title="deputy-director", location="on-campus"*}.

Let $errorTest(\phi, \varphi)$ represent an access request that satisfies $\phi \wedge \varphi$, where ϕ is a non-error constraint and φ is an error constraint. Because constraint solvers, including Z3, do not deal with error constraints in XACML, $errorTest(\phi, \varphi)$ first tries to solve the non-error constraint ϕ and then deals with the error constraint φ. Generally, the satisfiability of $\phi \wedge \varphi$ requires that φ should contain an attribute that does not appear in ϕ. If every attribute in the error constraint φ also appears in the non-error constraint ϕ, then the solution to the non-error constraint ϕ returned by the constraint solver, if exists, will have an attribute value for every attribute in the non-error constraint ϕ. Such a solution will never satisfy $\phi \wedge \varphi$ because ϕ requires each of its attributes (values) to have a correct category (type), whereas φ requires that at least one of its attributes (or values) should have an incorrect category (or type) or no value.

One of the major challenges of the proposed strong mutation approach for XACML policies is the transformation of collective reachability, necessity, and propagation constraints of $d(P, q) \neq d(P', q)$ into the form of $\phi \wedge \varphi$ with clear separation of non-error and error constraints. These constraints are much more complex than those in the coverage-based test generation [23]. Let $test(\delta)$ represent an access request that satisfies the constraint δ with mixed non-error and error predicates or clauses. The general constraint of $d(P, q) \neq d(P', q)$ is typically of the conjunctive normal form. The following is a generic example, where α is a normal or error constraint from the policy target, rt_i and rc_i are the target and condition of rule r_i.

$$\begin{aligned}&\alpha \wedge\\ &(\neg rt_1 \vee (rt_1 \wedge \neg rc_1) \vee error(rt_1) \vee (rt_1 \wedge error(rc_1))) \wedge\\ &\ldots\\ &(\neg rt_i \vee (rt_i \wedge \neg rc_i) \vee error(rt_i) \vee (rt_i \wedge error(rc_i))) \wedge\\ &\ldots\\ &(\neg rt_n \vee (rt_n \wedge \neg rc_n) \vee error(rt_n) \vee (rt_n \wedge error(rc_n)))\end{aligned}$$

Note that $\neg rt_i \vee (rt_i \wedge \neg rc_i) \vee error(rt_i) \vee (rt_i \wedge error(rc_i))$ means that rule r_i is not fired, i.e., $notFire(rt_i, rc_i) = notApplicable(rt_i, rc_i) \vee error(rt_i, rc_i)$. We realize $test(\delta)$ by applying both $normalTest(\phi)$ and $errorTest(\phi, \varphi)$. Each conjunction $\neg rt_i \vee (rt_i \wedge \neg rc_i) \vee error(rt_i) \vee (rt_i \wedge error(rc_i))$ is separated into four clauses: $\neg rt_i$, $rt_i \wedge \neg rc_i$, $error(rt_i)$, $rt_i \wedge error(rc_i)$. In addition to the handling of α, $test(\delta)$ uses the backtracking technique to find one solvable combination of the four clauses for all rules from r_1 to r_n. The combination is either a non-error constraint (e.g., $\alpha \wedge \neg rt_1 \wedge \ldots \neg rt_i \wedge \ldots \wedge rt_n \wedge rc_n$, assuming α is a normal constraint) or an error constraint with clear separation of non-error and error clauses (e.g.,$\alpha \wedge \neg rt_1 \wedge \ldots \neg rt_i \wedge \ldots \wedge error(rt_n)$). They are corresponding to $normalTest(\phi)$ and $errorTest(\phi, \varphi)$, respectively. In the worst-case scenario, the backtracking needs to explore 4^n possible combinations. This happens only when every rule uses a different set of attributes. In practice, we have not seen any XACML policy of this form.

To accelerate the backtracking process, it is important to take advantage of the logical relationships between different predicates and clauses. Consider the targets rt_i and rt_j of two rules that share the same set of attributes, e.g., *title="director"* in rule 1 and 2 and *title="deputy-director"* in rule 3 and 4 of the running example. rt_i may imply rt_j or $\neg rt_j$. rt_i (or $\neg rt_i$) implies that $error(rt_j)$ is not satisfiable and thus should not be considered. When $error(rt_i)$ is assumed, then rt_j or $\neg rt_j$ may not be considered. As such, the feasible combinations of rt_i and rt_j are significantly reduced. This is similar for rc_i and rc_j.

4.2 Reachability Constraint of Rules

A common function in our mutation-based test generation approach is the composition of reachability constraint that must be satisfied in order to trigger the evaluation of a given rule r_i in a policy. The reachability constraint depends on the policy target, all rules before rule r_i, and the rule-combining algorithm in the policy. If the rule-combining algorithm is *First-applicable*, r_i is not triggered unless all rules before it evaluate to N/A. If the rule-combining algorithm is *Deny-overrides* or *Permit-unless-deny*, r_i is triggered only if none of the deny rules before it is fired. If the rule-combining algorithm is *Permit-overrides* or *Deny-unless-permit*, r_i is triggered only if none of the permit rules before it is fired. Algorithm 1 describes how the reachability constraint is composed.

Algorithm 1: Compose reachability constraint of rule r_i

Function name: reachabilityConstraint
Input: Policy $P =< PT, RCA, [r_1, r_2, ..., r_n] >$, rule index $i(0 < i \leqslant n)$
Output: *constraint*

1 $constraint \leftarrow PT \vee error(PT)$;
2 **switch** *RCA* **do**
3 **case** *First-Applicable* **do**
4 **for** *each rule* $r_k = (rt_k, rc_k, re_k) \in [r_1, r_2, ..., r_{i-1}]$ **do**
5 $constraint \leftarrow constraint \wedge notApplicable(rt_k, rc_k)$;
6 **end**
7 **end**
8 **case** *Deny-overrides or Permit-unless-deny* **do**
9 **for** *each deny rule* $r_k = (rt_k, rc_k, Deny) \in [r_1, r_2, ..., r_{i-1}]$ **do**
10 $constraint \leftarrow constraint \wedge notFire(rt_k, rc_k)$;
11 **end**
12 **end**
13 **case** *Permit-overrides or Deny-unless-permit* **do**
14 **for** *each permit rule* $r_k = (rt_k, rc_k, Permit) \in [r_1, r_2, ..., r_{i-1}]$ **do**
15 $constraint \leftarrow constraint \wedge notFire(rt_k, rc_k)$;
16 **end**
17 **end**
18 **end**

For clarity, Algorithm 1 deals with an individual rule. As all rules are mutated for test generation, there is no need to apply it to every rule separately. Instead, it is used in an incremental fashion – the reachability constraint of rule r_{i+1} is composed from rule r_i and the reachability constraint of r_i.

5 TEST GENERATION FROM MUTANTS

In this section, we describe how to generate an access request q from an individual mutant P' of a given policy P such that $d(P, q) \neq d(P', q)$. Syntactically, the difference between P' and P is created by applying a mutation operator to a policy element in P. The semantic difference, however, also depends on other policy elements. It is captured by the collective reachability, necessity, and propagation constraints. The idea of test generation from a mutant is to compose and analyze these constraints, find one solution with a constraint solver, and convert the solution, if exists, into a request.

Our mutation-based test generation approach includes a significant number of test generation algorithms, e.g., at least one for the mutants of each mutation operator in Table 1. It would be tedious and unnecessary to elaborate on each of them. In this paper, we present the approach according to the fault types in Table 1. Although one fault type may have several mutation operators, the test generation algorithms for the mutation operators of the same fault type bear much similarity. Moreover, it is impossible to cover all fault types in this paper due to limited space. In the following, we only introduce the algorithms for mutants of incorrect policy target and incorrect rule effect and briefly discuss mutants of incorrect rule target/condition. We can formally prove that our test generation algorithms are sound: the resultant request q returned by each algorithm, if exists, always satisfies $d(P, q) \neq d(P', q)$. As an example, we will only present the proof of the algorithm for test generation from policy target mutants.

5.1 Policy Target Mutants

Given policy P, a policy target mutant P' is created by applying PTT (set Policy Target True), PTF (set Policy Target False), or RPTE (remove Parallel Target Element) to P's policy target without changing any other policy element. It simulates a faulty policy where the policy target is incorrect. Let $P =< PT, RCA, R >$ and $P' =< PT', RCA, R >$, where $R = [r_1, r_2, ..., r_n]$. To show the difference between P' and P, there is no reachability constraint because the target of a policy is always evaluated.

The necessity constraint is that the original policy target PT and the mutated policy target PT' evaluate to different results, i.e., $PT \neq PT'$. Different from the classical two-valued logic, here $PT \neq PT'$ implies six cases: $(PT \wedge \neg PT') \vee (\neg PT \wedge PT') \vee (PT \wedge error(PT')) \vee (\neg PT \wedge error(PT')) \vee (error(PT) \wedge PT') \vee (error(PT) \wedge \neg PT')$ because of the handling of error occurrence in XACML. It reduces to $(PT \wedge \neg PT') \vee (\neg PT \wedge PT')$ when error occurrences are not considered. It is simplified when a concrete mutation operator is applied. For PTT (i.e., PT' is true), $PT \neq PT'$ reduces to $\neg PT \vee error(PT)$. In Sample-PO-PTT, for example, the policy target is set to true. Test generation from this mutant will make the original policy target evaluate to false or error. For PTF (i.e., PT' is false), $PT \neq PT'$ reduces to $PT \vee error(PT)$.

Note that $PT \neq PT'$ does not guarantee that P' and P will result in different policy-level decisions. We need to propagate the difference of $PT \neq PT'$ to the policy-level, which depends on the rule-combining algorithm *RCA* and the rules. Algorithm 2 describes the procedure for generating a test from a policy target mutant, where $test(\delta)$ generates a request from the given constraint δ, which may or may not involve error occurrence. *fireRule* returns a request that activates the first firable permit or deny rule. *fireFirstApplicableRule* generates an access request that fires the first-applicable rule. *triggerRuleError* generates an access request that makes a rule evaluate to indeterminate.

If *RCA* is *Permit-unless-deny* or *Deny-unless-permit*, there is no propagation constraint. The test is generated from the necessity constraint (line 4). If *RCA* is *First-applicable*, the propagation constraint is that at least one rule fires (line 7) or, when PT or PT' is false, one of the rules evaluates to indeterminate (lines 8-13). If *RCA* is *Permit-overrides*, the propagation constraint is that one of the permit rules is fired (line 16), or when all permit rules are not-applicable (lines 20-23), a deny rule is fired (line 24), or, when PT or PT' is false, one of the rules is fired or evaluate to indeterminate (lines 25-36). The case of *Deny-overrides* is similar to that of *Permit-overrides* (i.e., permit is replaced with deny). It is thus not described in the algorithm. Considering the mutation operator used to create the mutant, the propagation constraint can be simplified. If it is PTT (i.e., PT' is true), $necessity \wedge \neg PT'$ is not satisfiable. Lines 11-13, 28-30, and 34-36 can be removed.

Algorithm 2: Generate an access request from a policy target mutant

Function name: policyTargetMutantTest
Input: Policy $P =< PT, RCA, R >$, mutant $P' =< PT', RCA, R >, R = [r_1, r_2, ..., r_n]$
Output: access request q

```
1  necessity ← PT ≠ PT';
2  switch RCA do
3      case Permit-unless-deny or Deny-unless-permit do
4          return test(necessity);
5      end
6      case First-Applicable do
7          q ← fireFirstApplicableRule(necessity, R);
8          if q = null then
9              q ← triggerRuleError(necessity ∧ ¬PT, R);
10         end
11         if q = null then
12             q ← triggerRuleError(necessity ∧ ¬PT', R);
13         end
14     end
15     case Permit-overrides do
16         q ← fireRule(necessity, R, Permit);
17         if q ≠ null then
18             return q;
19         end
20         nopermit ← true;
21         for each permit rule r_k = (rt_k, rc_k, Permit) ∈ R do
22             nopermit ← nopermit ∧ notApplicable(rt_k, rc_k);
23         end
24         q ← fireRule(necessity ∧ nopermit, R, Deny);
25         if q = null then
26             q ←
                fireFirstApplicableRule(necessity ∧ ¬PT, R);
27         end
28         if q = null then
29             q ←
                fireFirstApplicableRule(necessity ∧ ¬PT', R);
30         end
31         if q = null then
32             q ← triggerRuleError(necessity ∧ ¬PT, R);
33         end
34         if q = null then
35             q ← triggerRuleError(necessity ∧ ¬PT', R);
36         end
37     end
38 end
39 return q;
```

Theorem 1. Let $q = policyTargetMutantTest(P, P')$. If $q \neq null$, then $d(P, q) \neq d(P', q)$.

Proof: $q \neq null$ implies that $PT \neq PT'$ is satisfied (line 1 of the algorithm). We show that $d(P, q) \neq d(P', q)$ for each of the six cases of $PT \neq PT'$ according to Definition 2. The proof focuses on three cases because the others are symmetric.

(1) Suppose *RCA* is *Permit-unless-deny* or *Deny-unless-permit*. q only results from $test(necessity)$ in line 4, i.e., $test(PT \neq PT')$.

- $PT \wedge \neg PT'$: $d(P', q) = N/A$, but $d(P, q) = rca(RCA, R, q)$ is either *Permit* or *Deny*.
- $PT \wedge error(PT')$: $d(P, q)$ is either *Permit* or *Deny*, but $d(P', q) \in \{IP, ID, IDP\}$.
- $\neg PT \wedge error(PT')$: $d(P, q) = N/A$. However, $d(P', q) \in \{IP, ID, IDP\}$.

(2) Suppose *RCA* is *First-applicable*.

If q results from $fireFirstApplicableRule(PT \neq PT', R)$ in line 7, then $d(P, q) \neq d(P', q)$ for all cases of $PT \neq PT'$ as discussed below:

- $PT \wedge \neg PT'$: $d(P, q) = rca(RCA, R, q) \in \{Permit, Deny\}$, whereas $d(P', q) = N/A$.
- $PT \wedge error(PT')$: $d(P, q) = rca(RCA, R, q) \in \{Permit, Deny\}$. According to Definition 2, $d(P', q) = IP$ if $rca(RCA, R, q) = Permit$, and $d(P, q) = ID$ if $rca(RCA, R, q) = Deny$. So $d(P', q) \in \{IP, ID\}$. Therefore, $d(P, q) \neq d(P', q)$.
- $\neg PT \wedge error(PT')$: $d(P, q) = N/A$, whereas $d(P', q) \in \{IP, ID\}$.

If q results from $triggerRuleError(necessity \wedge \neg PT, R)$ in line 9, then $d(P, q) = N/A$. $rca(RCA, R, q) \in \{IP, ID\}$, and $PT \neq PT' \wedge \neg PT$ is satisfied. This implies $PT' \vee error(PT')$. In either case, $d(P', q) = rca(RCA, R, q) \in \{IP, ID\}$. This is similar for the case where q results from $triggerRuleError(necessity \wedge \neg PT', R)$ in line 12.

(3) Suppose *RCA* is *Permit-overrides*.

If q results from $fireRule(necessity, R, permit)$ in line 16, then there is a permit rule that is fired by q. Consider all the cases of $PT \neq PT'$.

- $PT \wedge \neg PT'$: $d(P, q) = rca(RCA, R, q) = Permit$, whereas $d(P', q) = N/A$.
- $PT \wedge error(PT')$: $d(P, q) = rca(RCA, R, q) = Permit$, whereas $d(P', q) = IP$.
- $\neg PT \wedge error(PT')$: $d(P, q) = N/A$, whereas $d(P', q) = IP$.

If q results from $fireRule(necessity \wedge nopermit, R, Deny)$ in line 24, then all permit rules are not applicable (lines 21-23) and a deny rule is fired (line 24). Consider all the cases of $PT \neq PT'$.

- $PT \wedge \neg PT'$: $d(P, q) = rca(RCA, R, q) = Deny$, whereas $d(P', q) = N/A$.
- $PT \wedge error(PT')$: $d(P, q) = rca(RCA, R, q) = Deny$, whereas $d(P', q) = ID$.
- $\neg PT \wedge error(PT')$: $d(P, q) = N/A$, whereas $d(P', q) = ID$.

If q results from $fireFirstApplicableRule(necessity \wedge PT, R)$ in line 26, then $rca(RCA, R, q) \in \{Permit, Deny\}$. $d(P, q) = N/A$ because of $\neg PT$. If PT' is true, then $d(P', q) = rca(RCA, R, q) \in \{Permit, Deny\}$. If $error(PT')$, then $d(P', q) \in \{IP, ID, IDP\}$. In either case, $d(P, q) \neq d(P', q)$. This is similar when q results from $fireFirstApplicableRule(necessity \wedge PT', R)$ in line 29.

If q results from $triggerRuleError(necessity \wedge \neg PT, R)$ in line 32, then $d(P, q) = N/A$ because of $\neg PT$. If PT' is true, then $d(P', q) \in \{IP, ID, IDP\}$. For either PT' or $error(PT')$, $d(P', q) = rca(RCA, R, q) \in \{IP, ID, IDP\}$. Thus, $d(P, q) \neq d(P', q)$. This is similar when q results from $triggerRuleError(necessity \wedge \neg PT', R)$ in line 35.

□

Theorem 1 demonstrates that Algorithm 2 is sound – a resultant request is sufficient for satisfying $d(P,q) \neq d(P',q)$. In fact, the algorithm is designed based on the sufficient and necessary condition of $d(P,q) \neq d(P',q)$. Consider *RCA = First-applicable* as an example. $fireFirstApplicableRule(necessity, R)$ would return an access request for a well-designed policy where the policy target is not in conflict with any rule target or rule condition (i.e., each rule is meant to be satisfiable under the given policy target). In reality, however, not all policies are well-designed. It is possible that $fireFirstApplicableRule(necessity, R)$ does not return a valid request. In this case, the algorithm continues to look for an alternative that satisfies $d(P,q) \neq d(P',q)$. If the necessary condition of $d(P,q) \neq d(P',q)$ is not addressed, we cannot conclude that P' is an equivalent mutant when the algorithm returns null. We have formalized the sufficient and necessary conditions for all types of XACML policy mutants in a separate paper. This paper will not prove how the test generation algorithms meet the necessary conditions. Note that, constraint satisfaction is in general a hard problem. The test generation algorithms in this paper are sound and complete only when the constraint solver used in the approach is sound and complete with respect to the constraints.

Let us consider the application of Algorithm 2 to *Sample-PO-PTT*. As mentioned before, the necessity constraint is that the original policy target, *department = "HR" or department = "IT"*, should evaluate to false or error. For the propagation constraint, the algorithm first checks on the permit rules one at a time. For the first permit rule, the combined constraint for test generation is: *((department = "HR" ∨ department = "IT") ∨ error (department = "HR" ∨ department = "IT")) ∧ title="director" ∧ location="on-campus"*. As this constraint is satisfiable, the call to $fireRule$ will return a valid request, such as {*department = "HR", title="director", location="on-campus"*}.

5.2 Rule Effect Mutant

Given policy P, a rule effect mutant P' is created by applying CRE (Change Rule Effect) to one of the rules in P without changing any other policy element. It simulates a faulty policy where a rule has an incorrect effect. Let $P =< PT, RCA, R >$ and $P' =< PT, RCA, R' >$, where $R = [r_1, ..., r_i, ..., r_n]$, $R' = [r_1, ..., r'_i, ..., r_n]$, and r_i and r'_i have the same rule target rt_i and rule condition rc_i, but different effects (one is deny and the other is permit). To show the difference between P' and P, we must reach r_i in P and r'_i in P' as described in Algorithm 1. The necessity constraint is $fire(rt_i, rc_i) \vee error(rt_i, rc_i)$. When *RCA=First-Applicable*, there is no propagation constraint. When *RCA=Permit-unless-deny* (or *Deny-unless-permit*), the propagation constraint requires that no other deny (or permit) rule should be fired. When *RCA=Permit-overrides*, the propagation constraint is that no other permit rule is fired if rule r_i is fired, or all other permit rules are not-applicable if rule r_i evaluates to indeterminate. Algorithm 3 describes how a test is generated from a rule effect mutant.

Theorem 2. Let q = *ruleEffectMutantTest*(P, P'). If $q \neq null$, then $d(P,q) \neq d(P',q)$.

□

Let us consider Sample-PO-CRE3 as an example, where the effect of rule 3 is changed from *Permit* to *Deny*. To reach rule 3, the policy target should evaluate to true or error, and none of the permit rules before rule 3 (rules 1 and 2) should be fired. Algorithm 3 will try to satisfy $constraint \wedge fire(rt_i, rc_i)$ (line 17), i.e., evaluate rule 3 to *Permit* in P, but *Deny* in P' and none of other permit rules after rule 3 fired (lines 18-20). The test can be generated from the following non-error constraint:

- Reachability: *(department = "HR" ∨ department = "IT") ∧¬ (title ="director" ∧ location="on-campus") ∧¬ (title = "director" ∧ location="off-campus")*
- Necessity: *(title="deputy-director" ∧ location="on-campus")*.

Algorithm 3: Generate an access request from a rule effect mutant

Function name: ruleEffectMutantTest
Input: Policy $P =< PT, RCA, [r_1, ...r_i, ..., r_n] >$, mutant $P' =< PT', RCA, [r_1, ...r'_i, ..., r_n] >$, $r_i =< rt_i, rc_i, re_i >$, $r'_i =< rt_i, rc_i, re'_i >$
Output: access request q

```
1  constraint ← reachabilityConstraint([r1, ..., rn], i);
2  switch RCA do
3      case Permit-unless-deny do
4          constraint ← constraint ∧ fire(rti, rci);
5          for each deny rule rk =< rtk, rck, Deny > in
             [ri+1, ..., rn] do
6              constraint ← constraint ∧ notFire(rtk, rck);
7          end
8          q ← test(newConstraint);
9      end
10     case First-Applicable do
11         q ← test(constraint ∧ fire(rti, rci));
12         if q = null then
13             q ← constraint ∧ error(rti, rci);
14         end
15     end
16     case Permit-overrides do
17         newConstraint ← constraint ∧ fire(rti, rci);
18         for each permit rule rk =< rtk, rck, Permit > in
             [ri+1, ..., rn] do
19             newConstraint ←
                 newConstraint ∧ notFire(rtk, rck);
20         end
21         q ← test(newConstraint);
22         if q = null then
23             newConstraint ← constraint ∧ error(rti, rci);
24             for each permit rule rk =< rtk, rck, Permit > in
                 [ri+1, ..., rn] do
25                 newConstraint ←
                     newConstraint ∧ notApplicable(rtk, rck);
26             end
27             q ← test(newConstraint);
28         end
29     end
30 end
31 return q;
```

- Propagation: ¬ *(title = "deputy-director" ∧ location = "off-campus")*.

The collective constraint is simplified as *(department = "HR" ∨ department = "IT") ∧ (title="deputy-director") ∧ (location="on-campus")*.

5.3 Rule Target/Condition Mutants

Given policy P, a rule target mutant P' is created by applying one of the following mutation operators to a rule target: RTT (set Rule Target True), RTF (set Rule Target False), and RPTE (Remove Parallel Target Element). It simulates a faulty policy where a rule has an incorrect rule target. Let $r_i =< rt_i, rc_i, re_i >$ and $r'_i =< rt'_i, rc_i, re_i >$. They have the same rule condition and effect, but different targets. To show the difference between P' and P, we must reach r_i in P and r'_i in P' as in Algorithm 1. The necessity constraint is $rt_i \neq rt'_i$. The propagation constraint depends on *RCA*.

Consider Sample-PO-RTT5, where the target of rule 5 is changed to true. The satisfiable non-error constraint for test generation is as follows:

- Reachability: the policy target is true and all rules before rule 5 are not applicable.
- Necessity: rule 5' original target is false.
- Propagation: rule 5's condition is true and no deny rule after rule 5 is fired.

The collective constraint is simplified as *department ∈ {"HR", "IT"} ∧ title ∉ {"deputy-director", "director"} ∧ job-class ∉ {"guest", "part-time", "intern", "contractor"} ∧ (location="off-campus")*.

In Sample-PO-RPTE5-1, *job-class="part-time"* as the second parallel element is removed from the target of rule 5. The necessity constraint is that the original rule target *job-class="guest" or job-class="part-time"* and the mutated target *job-class="guest"* evaluate to different results. This means that *job-class="part-time"*. The propagation constraint is the same as that for Sample-PO-RTT5. Thus, the satisfiable non-error constraint for test generation by Algorithm 7 is *department ∈ {"HR", "IT"} ∧ title ∉ {"deputy-director", "director"} ∧ (job-class="part-time") ∧ (location="off-campus")*.

A rule condition mutant created by applying one of the following mutation operators to a rule target: RCT (set Rule Condition True), RCF(set Rule Condition False), ANF (Add Not Function in condition), and RNF (Remove Not Function in condition). The test generation algorithm is similar to that for a rule target mutant, except for the additional reachability rt_i and the necessity constraint $rc_i \neq rc'_i$.

6 GENERATION OF OPTIMAL TEST SUITES

Let *mutationTest (P, P')* denote the test generated from P', a mutant of policy P. If P' is not equivalent to P, *mutationTest*(P, P') will return an access request to which P and P' produce different responses. If P is correct, then the test will kill P' (i.e., P' fails when it is exercised with the test). Note that, when a mutation operator is applied to a policy, it may yield a number of mutants. For example, the application of CRE (Change Rule Effect) produces n mutants, where n is the number of rules in the policy. Let Ω be a list of mutation operators and $M(P, \Omega)$ be the set of all non-equivalent mutants created by applying each mutant operator in Ω to policy P. The test suite $\{mutationTest(P, P') : P' \in M(P, \Omega)\}$ is the set of tests generated from all mutants in $M(P, \Omega)$. Its mutation score is 100% when all constraints are solvable. However, such a test suite is not necessarily optimal because a test generated from one mutant may kill another mutant. For example, a test generated from a CRE mutant may also kill the ANF and RNF mutants of the same rule.

Algorithm 4: Generate a test suite from all policy mutants

Function name: : SMT (or NO-SMT without lines 5 and 11)
Input: Policy $P =< PT, RCA, [r_1, r_2, ..., r_n] >$ and a set of mutation operators Ω
Output: A set of access requests Q
Variables: M is a mutant pool, OPS is a list of mutation operators, *constraint* is a logical constraint, q is an access request

```
1  Q ← ∅;
2  while Ω ≠ ∅ do
3      OPS ← select one or more mutation operators from Ω;
4      M ← a set of mutants created by all operators in OPS;
5      M ← M − kill(M, Q);
6      while M ≠ ∅ do
7          constraint ← constraint of one or more mutants;
8          q ← test(constraint);
9          if q ≠ null then
10             Q ← Q ∪ {q};
11             M ← M − kill(M, Q);
12         end
13     end
14 end
```

In this section, we discuss generation of optimal test suite that not only achieves a perfect mutation score, but has no duplicate tests. A test is considered duplicate if all mutants it kills can be killed by other tests. Due to the extra cost for test generation and test execution, duplicate tests significantly affect cost-effectiveness of testing. Our approach explores both static and dynamic strategies for the generation of an optimal test suite: (1) Static analysis of logical relationships between different types of mutants: We group mutants with some common satisfiable constraints (typically non-error constraints) and attempt to exploit these constraints first. If a common constraint is satisfiable, then only one test is generated for a group of mutants. (2) Dynamic checking: before the attempt to generate test from a new mutant, we run the mutant against the existing tests. If the mutant is killed, there is no need to generate a new test for the mutant.

Algorithm 4 describes the procedure for the generation of optimal test suite, where $kill(M, Q)$ denotes the set of mutants in M that are killed by test suite Q. We refer to the algorithm as **optimized Strong Mutation-based Testing (SMT)**, because it integrates the reachability, necessity, and propagation constraints of mutants and has no duplicate tests. Unless explicitly chosen by the user, Ω includes by default all the mutation operators in Table 1. Algorithm 4 groups mutants with common sufficient conditions that are identified through static analysis. It creates new mutants for one or mutation operators at a time (lines 3-4) and runs the mutants against the existing tests so as to avoid generating duplicate tests. The mutants that are already killed by the existing tests

are removed from the mutant pool (line 5). Then the algorithm composes the constraint for one or more compatible mutants (line 7) and attempts to generate a test from the constraint (line 8). If the constraint is solved, the solution is converted into a test and added to the test suite (lines 9-12), otherwise the mutant is considered an equivalent mutant. The algorithm also runs the new test against the current mutant pool. Mutants killed by the new test will be removed from the pool (line 11). The algorithm without optimization (lines 5 and 11) may also generate a test suite that achieves a perfect mutant score. It is called **Non-Optimized Strong Mutation-based Testing (NO-SMT)**.

Suppose n is the number of rules in the policy under test. Algorithm 4 deals with all mutants of the policy and, for each mutant, the reachability and propagation constraints require the traverse of almost all rules. The number of mutants is believed to be linear to n. Thus, the complexity of NO-SMT is $O(n2) \times O(Z3, n)$, where $O(Z3, n)$ stands for the complexity of *test(constraint)*, which relies on the Z3 constraint solver. As discussed in Section 4.1, the worst-case scenario can be $O(Z3) \times n^4$. $kill(M, Q)$ is to run the current test suite against the mutant pool. Executing a test with each mutant has a complexity of $O(n)$. So the complexity of $kill(M, Q)$ is $O(n^3)$, assuming the test suite size is linear to the policy size based on the observations in the experiments in the next section. The time complexity of $kill(M, q)$ within the inner while loop is similar. As such, the complexity of SMT is $O(n^3) \times O(Z3, n)$.

7 EXPERIMENTS

Our experiments aim to evaluate the performance of SMT (Algorithm 4) and NO-SMT (Algorithm 4 without lines 5 and 11) through various XACML policies and compare them with the existing methods from the perspectives of cost (e.g., test generation time and test suite size), effectiveness (e.g., mutation score) and cost-effectiveness (e.g., MKPT score). As this work is based on the current XACML 3.0 standard, our comparative study chooses three representative testing methods from those for XACML 3.0: rule coverage (RC), decision coverage (DC), and MC/DC [23]. Other testing methods include non-error decision coverage (NE-DC), non-error MC/DC (NE-MC/DC), rule pair coverage (PC), and permit/deny rule pair coverage (PD-PC). MC/DC subsumes NE-MC/DC, i.e., for a given policy, the MC/DC test suite is made up of the NE-MC/DC tests and certain error tests. DC subsumes NE-DC, i.e., the DC test suite is composed of the NE-MC/DC tests and certain error tests. PC and PD-PC are applicable only to the policies that contain both permit and deny rules. When applicable, they have had a poor record of fault detection[23]. Among the existing methods, MC/DC is the most effective as indicated by its highest mutation scores.

7.1 Experiment Setup

The XACML policies used in this paper include all the subjects in the relevant literature [23] as well as new examples. They are listed in Table 3. Policies 1-9 are those from the literature, demonstrating that the comparative study is meaningful. Policies 10-13 are new samples for further evaluating the testing methods. They are synthesized from the fedora policy by adding more rules and complex expressions.

Table 3: Subject Policies

No	Policy Name	RCA	#rules	LOC
1	Kmarket-blue	Deny-overrides	4	84
2	Kmarket-gold	Deny-overrides	4	58
3	Kmarket-silver	Deny-overrides	4	106
4	fedora	Deny-overrides	12	226
5	Conf	Permit-overrides	15	228
6	itrust	First-applicable	64	1,282
7	itrust5	First-applicable	320	6,402
8	itrust10	First-applicable	640	12,806
9	itrust20	First-applicable	1,280	25,602
10	fedora2	Deny-overrides	32	588
11	fedora3	Deny-overrides	212	2,748
12	fedora4	Deny-overrides	612	7,549
13	fedora5	Deny-overrides	312	4,727

Our experiments were performed on a 64bit Ubuntu laptop with 8th Generation Intel® Core™ i7-8550U Processor (1.80GHz 8MB) and 16.0GB DDR4 2400MHz. The open-source implementation together with all the new subject policies will be publicly available after the work has been accepted for publication.

7.2 Results

7.2.1 Test generation time and test suite size. Table 4 presents the test generation time of each testing method. It shows that all methods except SMT are scalable. NO-SMT is comparable to MC/DC, the most effective method in the existing study. It took SMT over 36 hours (129,656 seconds in Table 4) to complete the test generation for itrust10. Compared to NO-SMT, SMT's non-scalability is caused by the optimization that aims to identify and remove duplicate tests. We did not apply SMT to itrust20 because the previous study has shown that, for each of the existing methods, itrust20 has the same mutation score and MKPT score as itrust10. In essence, itrust5, itrust10, and itrust20 share the same policy structure and rule pattern but have different number of rules for the purposes of scalability study.

Table 5 presents the number of tests generated by each testing method. Even though SMT produces optimal test suites, its test suite sizes are not necessarily the smallest. For example, the size of each RC test suite is the number of rules. While it is the smallest, it is the least effective [23]. Generally, test suite size is not a good performance indicator of testing although it affects the test execution time. Nevertheless, SMT produces smaller test suites than MC/DC. The size of NO-SMT test suite is two-fold or three-fold of MC/DC. Compared to NO-SMT, SMT has a significant contribution to the removal of duplicate tests although it is very time-consuming. The removal rate, *(NO-SMT test suite size – SMT test suite size) / NO-SMT test suite size*, ranges from 55% to 68%.

7.2.2 Mutation score. Mutation score is the percentage of non-equivalent mutants killed by the test suite of a testing method. It has been commonly used as the primary indicator of testing effectiveness. In the existing study [23], RC did not kill most of the mutants, DC killed most of the mutants, and MC/DC killed the vast

Table 4: Time Generation Time (s)

Subject Policy	Testing Method RC	DC	MC/DC	NO-SMT	SMT
Kmarket-blue	.114	.264	.315	.600	.523
Kmarket-gold	.77	.236	.241	.419	.486
Kmarket-silver	.137	.371	.341	.605	.746
fedora	.396	.700	.865	1.87	3.07
Conf	.429	.427	.773	2.15	3.61
itrust	1.64	3.51	8.35	10.4	172
itrust5	8.71	24.4	64.8	48.1	15,876
itrust10	13.9	84.3	211	177	129,656
itrust20	45.9	563	696	710	-
fedora2	.847	1.29	1.94	4.53	14.9
fedora3	5.36	8.12	10.3	22.6	930
fedora4	16.4	37.5	40.2	73.6	14,725
fedora5	8.43	10.1	39.9	74.2	4,206

Table 5: Test Suite Sizes

Subject Policy	Testing Method RC	DC	MC/DC	NO-SMT	SMT
Kmarket-blue	4	10	11	17	7
Kmarket-gold	3	9	9	12	5
Kmarket-silver	5	13	13	19	8
fedora	12	25	30	61	21
Conf	15	16	25	74	25
itrust	64	66	197	387	196
itrust5	320	322	983	1,923	982
itrust10	640	642	1,965	3,843	1,964
itrust20	1,280	1,282	3,929	7,683	-
fedora2	32	45	70	161	60
fedora3	212	225	250	701	240
fedora4	612	625	650	1,901	640
fedora5	312	325	631	1,261	420

Table 6: Mutation Scores (%)

Subject Policy	Testing Method RC	DC	MC/DC	NO-SMT	SMT
fedora2	36.71	55.07	56.52	100	100
fedora3	27.62	51.13	51.46	100	100
fedora4	25.95	50.42	50.53	100	100
fedora5	47.83	67.67	74.17	100	100

majority of mutants. However, none of them was able to kill all mutants of all policies. Different from these methods, both SMT and NO-SMT have achieved a 100% mutation score for all policies. They can also determine whether or not a mutant is equivalent to the original policy, and if not, they produce an access request to which the mutant yields a different response.

In the existing study, MC/DC has obtained very high mutation scores, ranging from 97.7% to 100% [23]. As shown in Table 6, however, MC/DC's mutation scores can be as low as 50% for several new policies – it only kills about half of the non-equivalent mutants. At first sight, this is discouraging because the low mutation scores indicate that MC/DC is quite ineffective. An in-depth examination reveals that the large number of live mutants were all created by four mutation operators RTF, RCF, RER, and RPTE. These mutants are non-equivalent because SMT and NO-SMT have successfully generated access requests from each of them. MC/DC has killed all non-equivalent mutants created by all other mutation operators. As such, it can be inaccurate or misleading to use mutation scores as the key indicator of testing effectiveness.

Table 7: MKPT Scores

Subject Policy	Testing Method RC	DC	MC/DC	NO-SMT	SMT
Kmarket-blue	5.5	2.6	2.45	1.58	3.86
Kmarket-gold	5.67	2.22	2.22	1.75	4.2
Kmarket-silver	5.2	2.46	2.46	1.68	4
fedora	4.67	2.96	2.57	1.42	4.14
Conf	4.6	4.44	3.64	1.22	3.64
itrust	3	3.97	2.28	1.16	2.3
itrust5	3	3.99	2.28	1.17	2.28
itrust10	3	4	2.28	1.16	2.28
itrust20	3	4	2.28	1.16	-
fedora2	2.38	2.53	1.67	1.28	3.45
fedora3	1.21	2.11	1.91	1.32	3.86
fedora4	1.07	2.03	1.96	1.33	3.94
fedora5	2.43	3.3	1.87	1.26	3.78

7.2.3 MKPT score. MKPT score has been used to measure cost-effectiveness [23]. It is defined as the total number of killed mutants divided by the total number of tests. Table 7 shows the MKPT scores of each testing method. While NO-SMT and SMT both have perfect mutation scores, the test suite size of NO-SMT is two-fold or three-fold of SMT. Thus, the MKPT scores of SMT are two-fold or three-fold of NO-SMT. However, we cannot draw the conclusion that SMT is more cost-effective because SMT is not scalable from the perspective of test generation time. Note that even though SMT test suites are optimal, it does not mean they have the lowest MKPT scores. In fact, RC has the highest MKPT scores for several policies even though it is the least effective in terms of mutation scores. Compared to MC/DC, SMT has higher MKPT scores (except for Conf and intrustX), whereas NO-SMT has lower MKPT scores.

7.2.4 Summary. NO-SMT and SMT can kill all non-equivalent mutants, i.e., reveal all hypothesized faults in a given policy. Compared to the existing methods, however, they either produce larger test suites or require more time for test generation. While test generation time, mutation score, and MKPT score are all meaningful performance metrics, they should not be used alone for the measurement of testing effectiveness and cost-effectiveness. SMT is a viable method for thoroughly testing small policies and for comparing other testing methods to the optimal test suite. NO-SMT is scalable for dealing with all hypothesized faults of large policies. Because missing the chance of finding an access control fault can

lead to serious security incidents, effectiveness of access control testing must be given a priority. Thus, both NO-SMT and SMT are important for revealing access control faults. To achieve a perfect mutation score for a large policy, a scalable cost-effective approach is to combine MC/DC with the application of NO-SMT to the mutation operators whose mutants are not always killed by MC/DC. It is more effective than MC/DC (i.e., higher mutation score) and more cost-effective than NO-SMT (i.e., higher MPKT score).

8 CONCLUSIONS

We have presented the approach to automated strong mutation testing of XACML policies. It relies on the composition and solving of reachability, necessity, and propagation constraints that capture the semantic difference between a given policy and each of its mutants. Although constraint solving in general is a hard problem, the implementation of our approach has successfully generated mutation-based test suites for all the sample policies publicly available so far and nine new examples. These test suites can achieve a perfect mutation score and reveal all hypothesized faults. Thus, the mutation-based approach is more effective than the existing methods. It can also help improve other testing methods by determining whether or not a mutant is equivalent to its original policy, i.e., whether or not a syntactical difference in a mutant is a sematic fault. Nevertheless, the mutation-based approach is not necessarily the most cost-effective when test generation time and test suite size (or test execution time) are taken into consideration. For high assurance of large policies, a scalable yet more cost-effective approach is to enhance the MC/DC method by applying the mutation-based method (e.g., NO-SMT) to the mutation operators whose mutants are not killed by MC/DC. It would result in a higher mutation score than MC/DC and a higher MPKT score than NO-SMT.

In this paper, mutants are created by applying an individual mutation operator to make a single change to a given policy. They simulate various faults that likely occur in XACML policies. Using such first-order mutants has been the common practice in the field of mutation testing. Empirical studies have shown that program mutants are indeed similar to real faults in software for the purposes of evaluating software testing techniques [13]. A real fault might be a combination of single changes in a number of first-order mutants. The test generated from a first-order mutant would likely fail the program with a composite fault, and thus reveal the composite fault. We believe this is also the case for policy testing.

Our future work will investigate test generation from higher-order mutants that are created by applying more than one mutation operators or one mutation operator more than once. When debugging a program with multiple bugs, we often try to locate and fix the first one. Likewise, test generation from higher order mutants may start from the first mutated element and obtain its reachability and necessity constraints. The main challenge, however, is that the propagation constraint depends on other mutated elements.

ACKNOWLEDGMENTS

This work was supported in part by US National Science Foundation (NSF) under grant CNS 1954327 and grant CNS 1618229.

REFERENCES

[1] 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)*. 842–849.
[2] 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 Conf. on Web Information Systems and Technologies (WEBIST)*. 155–160.
[3] A. Bertolino, S. Daoudagh, F. Lonetti, and E Marchetti. 2013. XACMUT: XACML 2.0 mutants generator. In *Proc. of 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 28–33.
[4] A. Bertolino, Lonetti F., and Marchetti E. 2010. Systematic XACML request generation for testing purposes. In *EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*.
[5] A. Bertolino, Le Traon Y., F. Lonetti, E. Marchetti, and T. Mouelhi. 2014. Coverage-based test cases selection for XACML policies. In *2014 IEEE Seventh International Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*. 12–21.
[6] 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)*.
[7] R.A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* 11, 4 (1978), 34–41.
[8] R.A. DeMillo and A. J. Offut. 1991. Constraint-based automatic test data generation. *IEEE Trans. on Software Engineering* 17, 9 (September 1991), 900–910.
[9] R.A. DeMillo and A. J. Offut. 1993. Experimental results from an automatic test case generator. *ACM Trans. on Software Engineering Methodology* 2, 2 (1993), 109–127.
[10] S.J. Fan Chiang, D. Chen, and D. Xu. 2016. Conformance testing of Balana: An open source implementation of the XACML3.0 standard. In *Proc. of the 28th International Conf. on Software Engineering and Knowledge Engineering (SEKE'16)*.
[11] W. Howden. 1982. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering* 8, 4 (1982), 371–379.
[12] Y. Jia and M Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE Trans. on Software Engineering* 37, 5 (2010), 649–678.
[13] 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.
[14] S. Khamaiseh, P. Chapman, and D. Xu. 2018. Model-based testing of obligatory ABAC systems. In *Proc. of the 18th International Conference on Software Quality, Reliability and Security (QRS'18)*. 405–413.
[15] Y. Li, Y. Li, L. Wang, and 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)*.
[16] E. Martin and T. Xie. 2006. Automated test generation for access control policies. In *Supplemental Proc. of ISSRE*.
[17] 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.
[18] 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.
[19] OASIS. 2013. *eXtensible Access Control Markup Language (XACML) Version 3.0*. http://www.oasisopen.org/committees/xacml/.
[20] A. J. Offut. 2011. A mutation carol: Past, present and future. *Information and Software Technology* 53, 10 (2011), 1098–1107.
[21] 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* 9, 1 (2017), 73–91.
[22] D. Xu, N. Shen, and Y. Zhang. 2015. Detecting incorrect uses of combining algorithms in XACML 3.0 policies. *International Journal of Software Engineering and Knowledge Engineering* 25, 9/10 (2015), 1551–1571.
[23] D. Xu, R. Shrestha, and N Shen. 2018. Automated coverage-based testing of XACML policies. In *Proc. of the 23rd ACM Symposium on Access Control Models and Technologies (SACMAT'18)*. 3–14.
[24] D. Xu, Z. Wang, S. Peng, and N. Shen. 2016. Automated fault localization of XACML policies. In *Proc. of the 21st ACM Symposium on Access Control Models and Technologies (SACMAT'16)*. 137–147.
[25] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. 2010. Test generation via dynamic symbolic execution for mutation testing. In *Proc. of the 2010 IEEE International Conference on Software Maintenance (ICSM'10)*.