Towards a Theory on Testing XACML Policies

Dianxiang Xu
University of Missouri-Kansas City
Kansas City, Missouri, USA
dxu@umkc.edu

Roshan Shrestha
Boise State University
Boise, Idaho, USA
roshanshrestha@u.boisestate.edu

Ning Shen
Boise State University
Boise, Idaho, USA
ningshen@u.boisestate.edu

Yunpeng Zhang
University of Houston
Houston, Texas, USA
yzhan226@central.uh.edu

ABSTRACT

Policy testing is an important means for quality assurance of access control policies. Experimental studies on the testing methods of XACML policies have shown their varying levels of effectiveness. However, there is a lack of explanation for why they are unable to detect certain types of faults. It is unclear what is essential to the fault detection capability. To address this issue, we propose a theory on policy testing by formalizing the fault detection conditions with respect to a comprehensive fault model of XACML policies. The detection condition of a policy fault, composed of the reachability, necessity, and propagation constraints, is sufficient and necessary for revealing the fault. The formalized fault detection conditions can qualify the inherent strengths and limitations of testing methods. We have applied the formalization to the qualitative evaluations of five testing methods for the current version of the XACML standard. The results show that, for each method, there are certain types of faults that can always or never be revealed, while the detection of other faults may depend on the particular policy structure.

CCS CONCEPTS

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

KEYWORDS

access control, XACML, policy testing, mutation testing

ACM Reference Format:
Dianxiang Xu, Roshan Shrestha, Ning Shen, and Yunpeng Zhang. 2022. Towards a Theory on Testing XACML Policies. In *Proceedings of ACM Symposium on Access Control Models and Technologies (SACMAT'22).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3532105.3535031

1 INTRODUCTION

Policy testing has emerged as an important approach to security analysis of access control policies [1]-[4][11]-[15][18][19]. It aims at executing the policy under test with a test suite (i.e., a set of test cases) and checks if the policy produces a correct result for each test. A test case consists of a test input (i.e., access request) and an oracle value (i.e., correct response in terms of the actual access control requirement). It fails if the policy's actual response to the request is different from the oracle value. The failure implies the presence of policy faults (bugs). If none of the tests fails, however, one cannot assert that the policy is bug-free. To evaluate a testing method, the common approach is experimental mutation analysis. It creates a set of mutants of a given correct policy by seeding faults into the policy according to a fault model (i.e., typical bugs). The test cases produced by a testing method are then executed against all mutants. A mutant is said to be killed if at least one test fails. A live mutant not killed by any test is either faulty or functionally equivalent to the original policy (called equivalent mutant). The fault detection capability of the testing method is measured by its mutation score, i.e., the ratio between the number of mutants killed and the total number of non-equivalent mutants.

As an industry standard for access control policies used in major identity management products (e.g., Oracle Identity Manager and WSO2 Identity Server), XACML (eXtensible Access Control Markup Language) [16] has been the main subject of policy testing research. Mutation experiments using sample policies have shown that the existing testing methods have varying fault detection capabilities. The mutation scores range from 30% to 60% for Cirg [22], 75% to 79% for Targen [3] [13], 75% to 96% for X-CREATE [2], 62% to 98% for the rule-coverage-based test selection [4], 37% to 93% for XPTester [12], 50% to 63.6% for the rule-coverage (RC) testing [18], 62.5% to 96.6% for the decision-coverage (DC) testing [18], and 97.7% to 100% for the modified-condition/decision coverage (MC/DC) testing [18]. Recent work has found several XACML policies for which the mutation scores can be as allow as 26% for the RC method and 50% for the DC and MC/DC methods [19]. It remains unclear why the existing methods are unable to reveal certain types of faults. Although mutation experiments are useful for empirical evaluations, they do not address the fundamental question: what is essential to the fault detection capability of policy testing?

This paper presents a theory on testing XACML policies [16]. Unlike the existing work that relies on experiments, the proposed theory builds upon the formalization of fault detection conditions. A testing method can find a fault if and only if its tests can satisfy the detection condition. It is feasible to measure the inherent strengths and limitations of testing methods, independent of policy examples.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SACMAT'22, June 08–10, 2022, New York City, NY
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9357-7/22/06…$15.00
https://doi.org/10.1145/3532105.3535031

The contribution of this paper is twofold. First, we present a complete formalization of the sufficient and necessary fault detection conditions for all fault types in a well-studied fault model [18][19] of the current XACML standard. The fault detection condition consists of the reachability, necessity, and propagation constraints that any test must satisfy in order to reveal the fault in a given policy. Although the notions of reachability, necessity, and propagation constraints originate from program mutation testing or constraint-based software testing [6][7][8], there is a lack of formal treatment of these constraints. The problem with propagation constraints of software is known to be intractable [7][8] because of the explosion of program execution paths. As demonstrated in this paper, however, the unique features of XACML make it feasible to formalize the reachability, necessity, and propagation constraints of XACML policies in a three-valued logic (*true*, *false*, and *Error*). Understanding these constraints and thus fault detection conditions is essential to the design of a highly effective policy testing method.

Second, we apply the formalized fault detection conditions to the qualitative evaluation of five recent testing methods for XACML v3.0 policies: rule coverage (RC), decision coverage (DC), non-error decision coverage (NE-DC), modified decision/condition coverage (MC/DC), and non-error MC/DC (NE-MC/DC) [18]. The empirical studies have demonstrated that these methods outperform previous approaches [18]. In this paper, we evaluate them by analyzing whether or not they satisfy the fault detection condition of each fault type. This does not require the experimentation with specific policies, although we may use a running example for illustration purposes. The results of the qualitative evaluation show that there are certain types of faults that can always or never be revealed by each of the five methods, while the detection of other types of faults may depend on the particular policy structure. For example, the RC method cannot kill many types of mutants – this explains why its mutation scores are mostly low in the existing experiments.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes XACML policies and mutation operators. Section 4 presents the formalization of fault detection conditions. Section 5 evaluates the fault detection capabilities of five testing methods with the formalized fault detection conditions. Section 6 concludes this paper.

2 RELATED WORK

The existing approaches to testing XACML policies fall into two categories: model-based testing that derives tests from models and policy-based testing that produces test inputs directly from the policy under test. 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. [17] 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. [11] proposed a model-based testing method for obligatory Attribute-Based Access Control (ABAC) systems, where access control policies are implemented in XACML. Although the above methods have both involved XACML, the system implementation tested by the model-based approach may or may not rely on XACML. How to build effective test models, however, remains a critical challenge.

This paper is more related to the work that generates test inputs from the XACML policy under test. It has commonly used mutation analysis of XACML policies to measure testing effectiveness with mutation scores. Cirg [14] generates test inputs from counterexamples produced by the model checker Margrave 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. Targen [13] derives test inputs to satisfy all the possible combinations of truth-values of the attribute id-value pairs found in a given policy. 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]. They have also developed other test selection strategies, such as Simple Combinatorial and Incremental XPT [1]. Bertolino et al. [4] 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. XPTester uses symbolic execution technique to generate test inputs from XACML policies [12]. It converts the policy under test into a semantically equivalent C Code Representation (CCR) and symbolically executes CCR to create test inputs. The above methods are all based on earlier versions of XACML (1.0 or 2.0).

XPA (XACML Policy Analyzer) [18] offers various coverage-based testing methods for XACML v3.0 policies. They include rule-coverage (RC), decision coverage (DC), modified condition/decision coverage (MC/DC), non-error DC, and non-error MC/DC. The DC and MC/DC method can generate error tests, which are syntactically valid requests, but make decision expressions evaluate to error occurrence and result in an indeterminate decision. They are different from those tests using out-of-range attribute values [2], which make a logic expression evaluate to false, rather than error. XPA also provides a method for generating test cases from the policy mutants [19]. Different from the mutation analysis, this paper formalizes fault detection conditions of policy mutants, which opens the door for qualitative evaluation of testing methods.

3 XACML POLICIES AND POLICY MUTATION

3.1 XACML Policies

To avoid excessive formal notations, this paper focuses on policies, rather than policy sets. 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 rule or policy specifies the set of requests to which the rule or policy 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 [19] 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 rules 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", location="on-campus", Permit*>
R4: <*title="deputy", 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", 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.

Let ω be a decision expression (policy target, rule target, or rule condition) and q be an access request. 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) with respect to q, respectively. This is based on a three-valued logic because each predicate or logical expression has three possible outcomes. Per the XACML 3.0 standard specification [16], 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 follows:

$$d(r,q) = \begin{cases} Permit & \text{if } re = Permit \wedge rt \wedge rc \\ Deny & \text{if } re = Deny \wedge rt \wedge rc \\ N/A & \text{if } \neg rt \vee (rt \wedge \neg rc_i) \\ ID & \text{if } re = Deny \wedge \\ & (Error(rt) \vee rt \wedge Error(rc)) \\ IP & \text{if } re = Permit \wedge \\ & (Error(rt) \vee rt \wedge Error(rc)) \end{cases}$$

Table 1: Decision Table or Permit-overrides

	Permit	*Deny*	*N/A*	*ID*	*IP*	*IDP*
Permit	Permit	Permit	Permit	Permit	Permit	Permit
Deny	Permit	Deny	Deny	Deny	IP	IDP
N/A	Permit	Deny	N/A	ID	IP	IDP
ID	Permit	Deny	ID	ID	IDP	IDP
IP	Permit	IP	IP	IDP	IP	IDP

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 [19]:

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

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 rule-combining algorithm RCA in policy $P =< PT, RCA, R >$ aims at rendering a single decision by combining the decisions of all individual rules in R when the policy target PT is not false. RCA resolves the potential conflicting decisions of different rules. Given access request *{department = "HR", title="deputy-director", job-class="part-time", location="off-campus"}*, rules R4 and R6 in *Sample-PO* yield $Permit$ and $Deny$, respectively. The rule-combining algorithm *Permit-overrides* resolves the conflict by giving the priority to $Permit$, and thus the policy's decision is $Permit$. The main rule-combining algorithms in XACML 3.0 include *Permit-overrides (PO)*, *Deny-overrides (DO)*, *Permit-unless-deny (PUD)*, *Deny-unless-permit (DUP)*, *First-applicable (FA)*, *Ordered-permit-overrides*, and *Ordered-deny-overrides*. Their semantics are explained in the standard specification and can be formulated by decision tables [10]. For instance, Table 1 presents the decision table of *Permit-overrides*. Each entry represents a combined decision of the current policy decision (column) and the next rule decision (row). For example, if the current policy decision is *Deny* and the next rule evaluates to $Permit$, then the combined policy decision becomes $Permit$. The proofs of all theorems related to *Permit-overrides* rely on the decision table. This paper focuses on five rule-combining algorithms because *Ordered-permit-overrides* is similar to *Permit-overrides* and *Ordered-deny-overrides* is similar to *Deny-overrides*.

3.2 Mutation Operators

This paper formalizes fault detection conditions with respect to faults represented by policy mutants. Policy mutants are created by mutation operators, which modify a policy element of the given policy (i.e., mutation point) according to a fault type. We follow the current fault model and mutation operators of XACML 3.0 [18][19], as shown in Table 2. The fault types include incorrect policy (policy

Table 2: 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

Table 3: Sample Mutants of the *Sample-PO* Policy

Mutant Name	Mutated Element
Sample-PO-PTT	Policy target: true
Sample-PO-RPTE0-1	Policy target: department = "HR"
Sample-PO-CRE3	R3:<title="deputy", location= "on-campus", Deny>
Sample-PO-RTT5	R5: <true, location="off-campus", Deny>
Sample-PO-RPTE5-1	R5: <job-class="guest", location= "off-campus", Deny>
Sample-PO-RCT3	R3: <title="deputy", true, Permit>
Sample-PO-RER3	R3 is removed
Sample-PO-CRC-FA	RCA: First-applicable

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).

Each mutant is named after the original policy, the mutation operator, and the indices of the mutated element if applicable. Table 3 shows some sample 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 R3, which changes the effect from *Permit* to *Deny*.

4 FORMALIZATION OF FAULT DETECTION CONDITIONS

4.1 Fault Detection Conditions

Given a correct policy P and its mutant P' obtained by some mutation operator in Table 2, P' represents a policy fault if it is not an equivalent mutant. The fault detection condition (FDC) for revealing the fault in P' refers to the sufficient and necessary condition on access request q that makes P and P' produce different policy decisions, i.e., $d(P, q) \neq d(P', q)$. It consists of the following constraints:

- Reachability constraint: q must trigger the execution of the faulty policy element (i.e., mutation point) in P'.
- Necessity constraint: q must make the faulty policy element in P' and the original element in P evaluate to different intermediate results.
- Propagation constraint: q must make the different intermediate results propagated to the final decisions of P and P'.

P' is an equivalent mutant if the FDC is unsatifiable; otherwise, any request (test) satisfying the FDC can kill P', demonstrating that P' is different from P.

Consider mutant *Sample-PO-PTT* in Table 3, where the policy target is set to *true*. There is no reachability constraint because the policy target is always evaluated. The necessity constraint is that the original policy target is false, which makes the original policy evaluate to N/A. To propagate the difference to the policy level, the mutant's decision should not be N/A. Consider mutant *Sample-PO-CRE3*, where rule $R3$'s effect is changed. To reach rule $R3$, we may make the policy target evaluate to *true* and all rules before rule $R3$ evaluate to N/A. The necessity constraint is that rule $R3$ must be fired to obtain different rule-level decisions. To propagate the difference, no permit rule after rule $R3$ should be fired. Rule $R4$ is the only permit rule after rule $R3$. Putting the above constraints together, we have the following FDC:

(department="HR" $\vee$ *department="IT")* $\wedge$ //reachability
$\neg$ *(title="director"* $\wedge$ *location="on-campus")* $\wedge$
$\neg$ *(title="director"* $\wedge$ *location="off-campus")* $\wedge$
(title="deputy" $\wedge$ *location="on-campus")* $\wedge$ //necessity
$\neg$ *(title="deputy"* $\wedge$ *location="off-campus")* //propagation

It is simplified as *department* $\in$ *{"HR", "IT"}* $\wedge$ *(title="deputy")* $\wedge$ *(location="on-campus")*. Any test that satisfies this condition can kill the mutant.

In the following, we formalize the FDCs of all the mutants created by the mutation operators in Table 2. For each fault type with multiple mutation operators, we first deal with the general FDC and then apply it to the concrete mutation operators of the fault type. For example, a mutant with an incorrect policy target may result from the application of mutation operator PTT (set policy target true), PTF (set policy target false), or RPTE (remove parallel target element in policy target). The general FDC of incorrect policy target applies to all mutants of PTT, PTF, and RPTE. This not only facilitates the discussion, but also allows new mutation operators of the same fault type to be introduced in the future: the FDC of each new mutant is a simplified version of the general FDC. To avoid duplication, we will discuss the common fault detection constraints for all fault types beyond incorrect policy target.

For convenience, we also denote rule $r_i =< rt_i, rc_i, re_i >$ as $< rb_i, re_i >$, where rb_i, called rule body, consists of rt_i and rc_i. Per the XACML standard specification, $rb_i = rt_i \wedge rc_i$ when rule r_i fires; $\neg rbi = \neg rt_i \vee (rt_i \wedge \neg rc_i)$ when rule r_i is not applicable. $Error(rb_i) = Error(rt_i) \vee (rt_i \wedge Error(rc_i))$ when there is an error occurrence. Given an expression ω and its mutant ω', $\omega \neq \omega'$ means that ω and ω' evaluate to different results. There are six possible cases: $\omega \wedge \neg\omega'$, $\omega \wedge Error(\omega')$, $\neg\omega \wedge \omega'$, $\neg\omega \wedge Error(\omega')$, $Error(\omega) \wedge \omega'$, and $Error(\omega) \wedge \neg\omega'$. The theorems involving $\omega \neq \omega'$ are typically proven by dealing with all these cases.

4.2 FDC of Incorrect Policy Target

Let $P' =< PT', RCA, R >$ be a mutant of $P =< PT, RCA, R >$. P' has an incorrect policy target PT'. There is no reachability constraint because PT and PT' are always evaluated. The necessity constraint is $PT \neq PT'$. The propagation constraint depends on RCA. For *Permit-unless-deny* or *Deny-unless-permit*, there is no additional propagation constraint. The sufficient and necessary FDC is that PT and PT' evaluate to different results. This is formalized by Theorem 1.

Theorem 1. Suppose $P' =< PT', RCA, R >$ be a mutant of $P =< PT, RCA, R >$ and $RCA \in \{$*Permit-unless-deny*, *Deny-unless-permit*$\}$. $d(P, q) \neq d(P', q)$ if and only if $PT \neq PT'$ with respect to q.

Proof. (a) Sufficient condition: If $PT \neq PT'$, then $d(P, q) \neq d(P', q)$ for each of these six cases. We discuss three of them and others are symmetric.

If $PT \wedge \neg PT'$, then $d(P', q) = N/A$, but $d(P, q) = rca(RCA, R, q)$ is either *Permit* or *Deny*.

If $PT \wedge Error(PT')$, then $d(P, q)$ is either *Permit* or *Deny*, but $d(P', q)$ is either *ID* or *IP*.

If $\neg PT \wedge Error(PT')$, then $d(P, q) = N/A$, but $d(P', q)$ is either *ID* or *IP*.

(b) Necessary condition: $PT = PT'$ implies $d(P, q) = d(P', q)$ as they also have the same *RCA* and *R*. Therefore, $d(P, q) \neq d(P', q)$ implies $PT \neq PT'$. □

For *First-applicable*, the FDC is that $PT \neq PT'$ and at least one rule fires or, when PT or PT' is false, one rule evaluates to indeterminate. This is formalized by Theorem 2.

Theorem 2. Suppose $P' =< PT',$ *First-applicable*$, R >$ be a mutant of $P =< PT,$*First-applicable*$, R >$. $d(P, q) \neq d(P', q)$ if and only if $PT \neq PT' \wedge (\exists r_i \in R$ such that $rb_i \vee ((\neg PT \vee \neg PT') \wedge \exists r_i \in R$ such that $Error(rb_i)))$.

Proof. (a) Sufficient condition: Suppose $PT \neq PT' \wedge (\exists r_i \in R$ such that $rb_i \vee ((\neg PT \wedge \neg PT') \wedge \exists r_i \in R$ such that $Error(rb_i)))$ holds. $d(P, q) \neq d(P', q)$ holds for all the six cases of $PT \neq PT'$. We elaborate on three on them.

(1) $PT \wedge Error(PT')$: $(\neg PT \vee \neg PT')$ is false. "$(\exists r_i \in R$ such that $rb_i \vee ((\neg PT \wedge \neg PT') \wedge \exists r_i \in R$ such that $Error(rb_i)))$" reduces to "$\exists r_i \in R$ such that rb_i". Because *RCA* is *First-applicable*, $rca(RCA, R, q) \in \{Permit, Deny\}$ no matter whether there exists rule r_k before rule r_i that fires. According to Definition 2, $d(P, q) = rca(RCA, R, q) \in \{Permit, Deny\}$. $d(P', q)$ is simplified as follows:

$$d(P', q) = \begin{cases} rca & \text{if } rca \in \{N/A, ID, IP, IDP\} \\ IP & \text{if } rca = Permit \\ ID & \text{if } rca = Deny \end{cases}$$

where rca denotes $rca(RCA, R, q)$. Therefore, $d(P, q) \neq d(P', q)$.

(2) $PT \wedge \neg PT'$: $d(P, q) = rca(RCA, R, q)$ and $d(P', q) = N/A$ according to Definition 2. "$(\exists r_i \in R$ such that $rb_i \vee ((\neg PT \vee \neg PT') \wedge \exists r_i \in R$ such that $Error(rb_i)))$" reduces to "$\exists r_i \in R$ such that $rb_i \vee Error(rb_i)$". We prove that $rca(RCA, R, q) \neq N/A$. When there exists rule r_i such that rb_i, there are two cases: (i) no rule before r_i is fired. In this case, $d(r_i, q) = re_i$ according to Definition 1. $rca(RCA, R, q)$ will evaluate to re_i (either *Permit* or *Deny*). Therefore, $rca(RCA, R, q) \neq N/A$. (ii) rule r_k is the first rule before r_i that is fired. $rca(RCA, R, q)$ will evaluate to re_k (either *Permit* or *Deny*). Therefore, $rca(RCA, R, q) \neq N/A$. When there exists rule r_i such that $Error(rb_i)$, $rca(RCA, R, q) \neq N/A$ because $rca(RCA, R, q) = N/A$ only if all rules evaluate to N/A.

(3) $\neg PT \wedge Error(PT')$: $d(P, q) = N/A$. "$(\exists r_i \in R$ such that $rb_i \vee ((\neg PT \vee \neg PT') \wedge \exists r_i \in R$ such that $Error(rb_i)))$" reduces to "$\exists r_i \in R$ such that $rb_i \vee Error(rb_i)$". In either case, $rca(RCA, R, q) \neq N/A$. According to Definition 2, $d(P', q) \neq N/A$. So $d(P, q) \neq d(P', q)$.

(b) Necessary condition: Suppose $d(P, q) \neq d(P', q)$ and $PT \neq PT'$. We prove that $(\exists r_i \in R$ such that $rb_i \vee ((\neg PT \vee \neg PT') \wedge \exists r_i \in R$ such that $Error(rb_i)))$ holds for all six cases of $PT \neq PT'$. We elaborate on three of them.

(1) $PT \wedge Error(PT')$: $d(P, q) = rca(RCA, R, q)$, whereas

$$d(P', q) = \begin{cases} rca & \text{if } rca \in \{N/A, ID, IP, IDP\} \\ IP & \text{if } rca = Permit \\ ID & \text{if } rca = Deny \end{cases}$$

where $rca(RCA, R, q)$ is denoted by rca. If $rca(RCA, R, q) \in \{N/A, ID, IP, IDP\}$, then $d(P, q) = d(P', q)$, which is a contradiction. Thus, $rca(RCA, R, q) \in \{Permit, Deny\}$. This means that at least one rule fires, i.e., $\exists r_i \in R$ such that rb_i holds.

(2) $PT \wedge \neg PT'$: $d(P', q) = N/A$. $d(P, q) = rca(RCA, R, q) \neq N/A$. This means not all rules evaluate to N/A. In other words, there exists $r_i \in R$ such that rb_i or $Error(rb_i)$.

(3) $\neg PT \wedge Error(PT')$: $d(P, q) = N/A$. Because $d(P', q) \neq d(P, q)$, we have $rca(RCA, R, q) \neq N/A$. This implies that there exists $r_i \in R$ such that rb_i or $Error(rb_i)$. □

For *Deny-overrides*, the FDC is that $PT \neq PT'$ and one of the deny rules is fired, or a permit rule is fired but no deny rule evaluates to indeterminate, or, when PT or PT' is false, one rule is fired or evaluate to indeterminate. This is formalized by Theorem 3.

Theorem 3. Suppose $P' =< PT',$*Deny-overrides*$, R >$ be a mutant of $P =< PT,$*Deny-overrides*$, R >$. $d(P, q) \neq d(P', q)$ if and only if $PT \neq PT' \wedge (\exists r_i = (rb_i, Deny) \in R$ such that $rb_i \vee (\exists r_i = (rb_i, Permit) \in R$ such that $rb_i \wedge \neg \exists r_j = (rb_j, Deny) \in R$ such that $Error(rb_j))) \vee ((\neg PT \vee \neg PT') \wedge \exists r_i \in R$ such that $rb_i \vee Error(rb_i))$.

Proof. (a) Sufficient condition: We discuss three cases of $PT \neq PT'$.

(1) $PT \wedge Error(PT')$: Because $\neg PT \vee \neg PT'$ is false, $\exists r_i = (rb_i, Deny) \in R$ such that $rb_i \vee (\exists r_i = (rb_i, Permit) \in R$ such that $rb_i \wedge \neg \exists r_j = (rb_j, Deny) \in R$ such that $Error(rb_j))$. Thus, $rca(RCA, R, q) \in \{Permit, Deny\}$. $d(P, q) = rca(RCA, R, q) \in \{Permit, Deny\}$. According to Definition 2, $d(P', q) \in \{IP, ID\}$. Therefore, $d(P, q) \neq d(P', q)$.

(2) $PT \wedge \neg PT'$: $d(P', q) = N/A$. If $\exists r_i = (rb_i, Deny) \in R$ such that $rb_i \vee (\exists r_i = (rb_i, Permit) \in R$ such that $rb_i \wedge \neg \exists r_j = (rb_j, Deny) \in R$ such that $Error(rb_j))$, $d(P, q) = rca(RCA, R, q) \in \{Permit, Deny\}$. Therefore, $d(P, q) \neq d(P', q)$. If $\exists r_i \in R$ such that rb_i or $Error(rb_i)$, $d(P, q) = rca(RCA, R, q) \neq N/A$. Thus, $d(P, q) \neq d(P', q)$.

(3) $\neg PT \wedge Error(PT')$: $d(P, q) = N/A$, whereas:

$$d(P', q) = \begin{cases} rca & \text{if } rca \in \{N/A, ID, IP, IDP\} \\ IP & \text{if } rca = Permit \\ ID & \text{if } rca = Deny \end{cases}$$

If $\exists r_i = (rb_i, Deny) \in R$ such that $rb_i \vee (\exists r_i = (rb_i, Permit) \in R$ such that $rb_i \wedge \neg \exists r_j = (rb_j, Deny) \in R$ such that $Error(rb_j))$, $d(P', q) = rca(RCA, R, q) \in \{Permit, Deny\}$. Therefore, $d(P, q) \neq$

$d(P', q)$. If $\exists r_i \in R$ such that rb_i or $Error(rb_i)$, $rca(RCA, R, q) \neq N/A$. So $d(P', q) \neq N/A$.

(b) Necessary condition: Suppose $d(P, q) \neq d(P', q)$. For each of the six cases of $PT \neq PT'$, we prove that $(\exists r_i = (rb_i, Deny) \in R$ such that $rb_i \vee (\exists r_i = (rb_i, Permit) \in R$ such that $rb_i \wedge \neg\exists r_j = (rb_j, Deny) \in R$ such that $Error(rb_j)))) \vee ((\neg PT \vee \neg PT') \wedge \exists r_i \in R$ such that $rb_i \vee Error(rb_i))$.

(1) $PT \wedge Error(PT')$: $d(P, q) = rca(RCA, R, q)$. If $rca(RCA, R, q) \in \{N/A, ID, IP, IDP\}$, then $d(P, q) = d(P', q)$, which is a contradiction. Thus, $rca(RCA, R, q) \in \{Permit, Deny\}$. This means that $\exists r_i = (rb_i, Deny) \in R$ such that $rb_i \vee (\exists r_i = (rb_i, Permit) \in R$ such that $rb_i \wedge \neg\exists r_j = (rb_j, Deny) \in R$ such that $Error(rb_j))$.

(2) $PT \wedge \neg PT'$: $d(P', q) = N/A$. $d(P, q) = rca(RCA, R, q) \neq N/A$. This means not all rules evaluate to N/A. In other words, there exists $r_i \in R$ such that rb_i or $Error(rb_i)$. This subsumes $rca(RCA, R, q) \in \{Permit, Deny\}$. So $(\exists r_i = (rb_i, Deny) \in R$ such that $rb_i \vee (\exists r_i = (rb_i, Permit) \in R$ such that $rb_i \wedge \neg\exists r_j = (rb_j, Deny) \in R$ such that $Error(rb_j))) \vee (\exists r_i \in R$ such that rb_i or $Error(rb_i))$.

(3) $\neg PT \wedge Error(PT')$: $d(P, q) = N/A$. Because $d(P', q) \neq d(P, q)$, we have $rca(RCA, R, q) \neq N/A$. This implies that there exists $r_i \in R$ such that rb_i or $Error(rb_i)$. □

Since *Permit-overrides* and *Deny-overrides* are symmetric, this paper will present the relevant theorems for one of them. Now we discuss how the above FDCs apply to the incorrect policy target mutants created by the relevant mutation operators. When P' is a PTT mutant of P, i.e., PT' is always true. The necessity constraint $PT \neq PT'$ reduces to $\neg PT \vee Error(PT)$. When P' is a PTF mutant of P, i.e., PT' is always false, the necessity constraint reduces to $PT \vee Error(PT)$. For RPTE, it can be applied to either an AllOf or AnyOf clause. Consider $PT = c_1 \wedge ... \wedge c_{i-1} \wedge c_i \wedge c_{i+1} ... \wedge c_n$ and $PT' = c_1 \wedge ... \wedge c_{i-1} \wedge c_{i+1} ... \wedge c_n$ where c_i is removed. When PT is true, PT' is true. When PT' is false, PT is false. When PT' evaluates to error, PT also evaluates to error. Thus $PT \wedge (\neg PT' \vee Error(PT'))$ is false, $\neg PT \wedge (PT' \vee Error(PT'))$ becomes $\neg c_i \wedge c_1 \wedge ... \wedge c_{i-1} \wedge c_{i+1} ... \wedge c_n$. $Error(PT) \wedge (PT' \vee \neg PT')$ becomes $Error(c_i) \wedge \neg Error(c_j)$ for any $j \neq i$. Thus, the necessity constraint reduces to $\neg c_i \wedge c_1 \wedge ... \wedge c_{i-1} \wedge c_{i+1} ... \wedge c_n \wedge Error(c_i) \wedge \neg Error(c_j)$ for any $j \neq i$. It means that the removed clause evaluates to false while all other clauses evaluates to true or it evaluates to error while none of the other clauses evaluates to error. In essence, this is to capture the impact of the removed clause c_i. A special case is that $PT = c_1$ and c_1 is removed, it reduces to $\neg c_1 \vee Error(c_1)$. This is the same as the case of PTT mutant. Similarly, when RPTE removes c_i from the *AnyOf* clause $c_1 \vee ... \vee c_{i-1} \vee c_i \vee c_{i+1} ... \vee c_n$, the necessity constraint is simplified to capture the impact of the removed clause c_i.

4.3 Common Constraints for Other Fault Types

Before presenting the fault detection condition of other fault types, let us first discuss their common reachability and propagation constraints. To reach the rule-combining algorithm and individual rules of a policy, the policy target should not evaluate to false. Let $P' =< PT, RCA', R' >$, where the mutated element is either the rule-combining algorithm or a rule. When PT is true, $d(P, q) \neq d(P', q)$ if and only if $rca(RCA, R, q) \neq rca(RCA', R', q)$. When an error occurs in the evaluation of PT, $d(P, q) \neq d(P', q)$ if and only if $rca(RCA, R, q) \neq rca(RCA', R', q)$ and $< Permit, IP >$ and $< Deny, ID >$ are not the results of $< rca(RCA, R, q), rca(RCA', R', q) >$ or $< rca(RCA', R', q), rca(RCA, R, q) >$. So the common reachability constraint is $PT \vee Error(PT)$, and the common propagation constraint is $Error(PT) \rightarrow \neg(rca(RCA, R, q) = Permit \wedge rca(RCA', R', q) = IP) \wedge \neg(rca(RCA, R, q) = IP \wedge rca(RCA', R', q) = Permit) \wedge \neg(rca(RCA, R, q) = Deny \wedge rca(RCA', R', q) = ID) \wedge \neg(rca(RCA, R, q) = ID \wedge rca(RCA', R', q) = Deny)$. In the following, these constraints are assumed. Because they have covered the error occurrence of PT, we focus on $rca(RCA, R, q) \neq rca(RCA', R', q)$ when PT is true.

4.4 FDC of Incorrect Rule Effect

Suppose $P' =< PT, RCA, R' >$ is a mutant of $P =< PT, RCA, R >$ with an incorrect rule effect. Without loss of generality, we assume that rule r_i in P is $< rb_i, Permit >$, whereas the incorrect rule r_i in P' is $< rb_i, Deny >$. While the necessity constraint is that rule r_i in P and rule r_i in P' produce different rule-level decisions, the additional reachability and propagation constraints depend on *RCA*.

When *RCA* =*Permit-overrides*, the additional condition besides the aforementioned common constraints is that rule r_i is fired and no other permit rule is fired, or when rule r_i evaluates to indeterminate, all other permit rules are not-applicable. This is formalized by Theorem 4.

Theorem 4. *$rca(Permit\text{-}overrides, R, q) \neq rca(Permit\text{-}overrides, R', q)$ if and only if $rb_i \wedge (\neg rb_j \vee Error(rb_j))$ for any permit rule $r_j =< rb_j, Permit > (j \neq i) \vee Error(rb_i) \wedge \neq rb_j$ for any permit rule $r_j =< rb_j, Permit > (j \neq i)$.*

Proof. (a) Sufficient condition: Suppose $rb_i \wedge (\neg rb_j \vee Error(rb_j))$ for any permit rule $r_j =< rb_j, Permit > (j \neq i)$. Since no permit rule before r_i is fired, r_i is reached. It is fired because rb_i is true, and evaluates to its effect (i.e., *Permit* in P and *Deny* in P'). So $rca(Permit\text{-}overrides, R, q) = Permit$. Since no permit rule after r_i is fired, $rca(Permit\text{-}overrides, R', q) \neq Permit$. Thus, $rca(Permit\text{-}overrides, R, q) \neq rca(Permit\text{-}overrides, R', q)$.

Suppose $Error(rb_i) \wedge \neg rb_j$ for any permit rule $r_j =< rb_j, Permit > (j \neq i)$. Since no permit rule before r_i is fired, r_i is reached. It evaluates to *IP* in P and *ID* in P' because $Error(rb_i)$. If r_i is the last rule, then $rca(Permit\text{-}overrides, R, q) = IP$, and rca(Permit-overrides, $R', q) = ID$. Thus, $rca(Permit\text{-}overrides, R, q) \neq rca(Permit\text{-}overrides, R', q)$. If r_i is not the last rule, consider each rule after r_i. It cannot evaluate to *Permit* or *IP* because $\neg rb_j$ for any permit rule $r_j =< rb_j, Permit > (j \neq i)$. So each rule after r_i evaluates to N/A, *Deny*, or *ID*. $rca(Permit\text{-}overrides, R, q) \in \{IP, IDP\}$, whereas $rca(Permit\text{-}overrides, R', q) \in \{Deny, ID\}$.

(b) Necessary condition: Suppose $rca(Permit\text{-}overrides, R, q) \neq rca(Permit\text{-}overrides, R', q)$. $\neg rb_i$ should not hold, otherwise rule r_i evaluates to N/A and $rca(Permit\text{-}overrides, R, q) = rca(Permit\text{-}overrides, R', q)$. Thus, $rb_i \vee Error(rb_i)$. In the case of rb_i, $rca(Permit\text{-}overrides, R, q) \neq rca(Permit\text{-}overrides, R', q)$ implies that $(\neg rb_j \vee Error(rb_j))$ for any permit rule $r_j =< rb_j, Permit > (j \neq i)$, otherwise both evaluate to *Permit*. In the case of $Error(rb_i)$, rule r_i evaluates to *IP* in P and *ID* in P'. $rca(Permit\text{-}overrides, R, q) \neq rca(Permit\text{-}overrides, R', q)$ implies that each other permit rule should evaluate to N/A, i.e., $\neg rb_j$ for any permit rule $r_j =< rb_j, Permit > (j \neq i)$. □

When *RCA =Permit-unless-deny*, the additional fault detection constraint is that rule r_i is fired and no other deny rule is fired. This is formalized by Theorem 5.

Theorem 5. $rca(Permit\text{-}unless\text{-}deny, R, q) \neq rca(Permit\text{-}unless\text{-}deny, R', q)$ if and only if $rb_i \wedge (\neg rb_j \vee Error(rb_j))$ for any deny rule $r_j =< rb_j, Deny >$ $(j \neq i)$.

When *RCA =First-Applicable*, the additional fault detection constraint is that rule r_i is fired and no rule before r_i is fired, or rule r_i evaluates to indeterminate, all rules before r_i are not-applicable, and no rule after r_i is fired. This is formalized by Theorem 6.

Theorem 6. $rca(First\text{-}Applicable, R, q) \neq rca(First\text{-}Applicable, R', q)$ if and only if $rb_i \wedge (\neg rb_j \vee Error(rb_j))$ for any rule $r_j (j < i) \vee Error(rb_i) \wedge \vee rb_j$ for any rule $r_j =< rb_j, re_j > (j < i) \wedge (\neg rb_k \vee Error(rb_k))$ for any rule $r_k =< rb_k, re_k > (k > i)$.

4.5 FDC of Incorrect Rule Target/Condition

Suppose $P' =< PT, RCA, R' >$ is a mutant of $P =< PT, RCA, R >$, where the mutation point is the body (target or condition) of rule r_i, i.e., $< rb_i, re_i >$ in P and $< rb'_i, re_i >$ in P'. The reachability and propagation constraints depend on *RCA*. When *RCA =First-applicable*, the additional fault detection constraint is that none of the rules before rule r_i are fired, and the rule bodies rb_i and rb'_i evaluate to different results, and when one of rb_i and rb'_i is true, there should not exist rule r_j after r_i that is fired with the same effect as r_i, or when one of rb_i and rb'_i is false and the other is error, all rules before r_j evaluate to N/A, no rule after r_i is fired, and there exists rule r_s with different effect than r_i that evaluates to indeterminate and all rules between r_i and r_s evaluate to N/A or all rules after r_i evaluate to N/A. This is formalized by Theorem 7.

Theorem 7. $rca(First\text{-}Applicable, R, q) \neq rca(First\text{-}Applicable, R', q)$ if and only if $\neg rb_j \vee Error(rb_j)$ for any rule $r_j =< rb_j, re_j > (j < i) \wedge rb_i \neq rbi' \wedge ((rb_i \vee rb'_i) \wedge (\neg \exists r_j =< rb_j, re_j > (j > i)$ such that $(re_j = re_i) \wedge rb_i) \vee (\neg rb_i \wedge Error(rb'_i) \vee Error(rb_i) \wedge \neg rb'_i) \wedge \neg rb_j$ for any rule $r_j =< rb_j, re_j > (j < i) \wedge \neg \exists r_k =< rb_k, re_k > (k > i)$ such that rb_k holds $\wedge (\exists r_s =< rb_s, re_s > (s > i)$ such that $(re_l \neq re_i) \wedge Error(rb_s) \wedge \neg rb_t$ for any rule $r_t =< rb_t, re_t > (i < t < s) \vee \neg rb_u$ for any rule $r_u =< rb_u, re_u > (i < u)))$.

When *RCA =Permit-overrides*, none of the permit rules before rule r_i should be fired, otherwise rule r_i is not evaluated. The necessity constraint requires that the rule bodies rb_i and rb'_i evaluate to different results. When one of rb_i and rb'_i is true, none of the permit rules after rule r_i should be fired if rule r_i is a permit rule and none of the other rules should be fired if rule r_i is a deny rule. When one of rb_i and rb'_i is false and the other is error, all permit rules after rule r_i should be non-applicable and no deny rule is fired if rule r_i is a permit rule, and none of the other rules should be fired and no deny rule should occur an error if rule r_i is a deny rule.

When *RCA =Permit-unless-deny*, none of the deny rules before r_i should be fired, otherwise r_i is not evaluated. The necessity constraint requires that r_i should be a deny rule: if r_i is a permit rule, both $rca(Permit\text{-}unless\text{-}deny, R, q)$ and $rca(Permit\text{-}unless\text{-}deny, R', q)$ depend on the rules after r_i and thus have the same decision. The necessity constraint also requires that rb_i and rb'_i evaluate to different results but one of them is true. The propagation constraint requires that none of the other deny rules should be fired. This is formalized by Theorem 8.

Table 4: Instantiation of $rb_i \neq rb'_i \wedge (rb_i \vee rb'_i)$

Mutation Operator	Mutated Rule in P' (i.e., $rb'_i =$)	Instantiated version of $rb_i \neq rb'_i \wedge (rb_i \vee rb'_i)$
RTT	$< true, rc_i >$	$(\neg rt_i \vee Error(rt_i)) \wedge rc_i$
RTF	$< false, rc_i >$	$(rt_i \wedge rc_i) \vee (Error(rt_i) \wedge \neg rc_i)$
RCT	$< rt_i, true >$	$rt_i \wedge \neg rc_i$
RCF	$< rt_i, false >$	$rt_i \wedge rc_i$
ANF	$< rt_i, not\ rc_i >$	rt_i

Table 5: Instantiation of $\neg rb_i \wedge Error(rb'_i) \vee Error(rb_i) \wedge \neg rb'_i$

Mutation Operator	Mutated Rule in P' (i.e., $rb'_i =$)	Instantiated version of $\neg rb_i \wedge Error(rb'_i) \vee Error(rb_i) \wedge \neg rb'_i$
RTT	$< true, rc_i >$	$(\neg rt_i \wedge Error(rc_i)) \vee (Error(rt_i) \wedge \neg rc_i)$
RTF	$< false, rc_i >$	$(Error(rt_i) \wedge \neg rc_i)$
RCT	$< rt_i, true >$	Not satifiable
RCF	$< rt_i, false >$	$rt_i \wedge Error(rc_i)$
ANF	$< rt_i, not\ rc_i >$	Not satifiable

Theorem 8. $rca(Permit\text{-}unless\text{-}deny, R, q) \neq rca(Permit\text{-}unless\text{-}deny, R', q)$ if and only if $re_i = Deny \wedge rb_i \neq rb'_i \wedge (rb_i \vee rb'_i) \wedge (\neg rb_j \vee Error(rb_j))$ for any deny rule $r_j =< rb_j, Deny >$ $(j \neq i)$.

In the above general FDCs, the key expressions about the mutation point are $rb_i \neq rb'_i \wedge (rb_i \vee rb'_i)$ and $\neg rb_i \wedge Error(rb'_i) \vee Error(rb_i) \wedge \neg rb'_i$. They can be instantiated for each mutant type of incorrect rule target and incorrect rule condition. Tables 4 and 5 show their instantiated versions for the mutants corresponding to mutation operators RTT (Rule Target True), RTF (Rule Target False), RCT (Rule Condition True), RCF (Rule Condition False), and ANF (Add Not Function in rule condition).

For each mutant of incorrect rule condition, the satisfaction of rule target rt_i is also a reachability constraint: the mutation point rc'_i is evaluated only when rt_i evaluates to $true$. Thus, rb_i becomes $rt_i \wedge rc_i$ and $\neg rb_i$ becomes $rt_i \wedge \neg rc_i$. For a RCT mutant where $rb_i =< rt_i, rc_i >$ and $rb'_i =< rt_i, true >$, the constraint $\neg rb_i \wedge Error(rb'_i) \vee Error(rb_i) \wedge \neg rb'_i$ is not satisfiable because $Error(rb'_i) = Error(rt_i)$ is impossible (actually rt_i is true) when $\neg rb_i$ (i.e., $rti \wedge \neg rc_i$), and $\neg rb'_i$ (i.e., $rt'_i \wedge \neg rc'_i$) is impossible (actually rb'_i evaluates to error) when $Error(rb_i) = Error(rt_i)$. In this case, the fault detection condition is simplified by removing the entire sub-constraint of $\neg rb_i \wedge Error(rb'_i) \vee Error(rb_i) \wedge \neg rb'_i$. For an RCF mutant where $rb_i =< rt_i, rc_i >$ and $rb'_i =< rt_i, false >$, rb'_i is false whenever rb_i evaluates to true (i.e., $rt_i \wedge rc_i$) or error (i.e., $rt_i \wedge Error(rc_i)$).

For an ANF mutant where $rb_i =< rt_i, rc_i >$ and $rb'_i =< rt_i, not\ rc_i >$. Whenever rt_i is true (reachability constraint), one of rb_i and rb'_i is true and the other is false. The necessity constraint of the mutation point is met. $(\neg rb_i \wedge Error(rb'_i) \vee Error(rb_i) \wedge \neg rb'_i)$ is not satisfiable because rb'_i evaluates to indeterminate if and only if rb_i evaluates to indeterminate. Thus, this sub-constraint can be removed.

4.6 FDC of Missing Rule

Suppose $P' =< PT, RCA, R' >$ is a mutant of $P =< PT, RCA, R >$, where rule r_i in R is missing in R'. Let $R =< r_1, ..., r_{i-1}, r_i, r_{i+1}, ...r_n >$, and $R' =< r_1, ..., r_{i-1}, r_{i+1}, ..., r_n >$. When RCA =*Permit-overrides*, no permit rule before rule r_i should be fired to reach rule r_i in P. To make rule r_i take effect, r_i should not evaluate to N/A. (1) When rule r_i in P is fired and its decision is re_i. If $re_i = Permit$, then the propagation constraint requires that no permit rule after rule r_i should be fired. Together with the reachability constraint, it means that no other permit rule should be fired. If $re_i = Deny$, then no permit rule after rule r_i should be fired and none of the other deny rules should be fired. Together with the reachability constraint, it means that no other rule should be fired. (2) When rule r_i evaluates to indeterminate and its decision is $I(re_i)$. If $re_i = Permit$, all other permit rules should evaluate to N/A. If $re_i = Deny$, all other rules should evaluate to N/A. Formally, we have Theorem 9.

Theorem 9. Let $R = [r_1, ..., r_{i-1}, r_i, r_{i+1}, ..., r_n]$, $R' = [r_1, ..., r_{i-1}, r_{i+1}, ..., r_n]$. $rca(Permit\text{-}overrides, R, q) \neq rca(Permit\text{-}overrides, R', q)$ if and only if $(rb_i \wedge (re_i = Permit) \wedge (\neg rb_j \vee Error(rb_j))$ for any permit rule $r_j =< rb_j, Permit > (j \neq i)) \vee (rb_i \wedge (re_i = Deny) \wedge (\neg rb_j \vee Error(rb_j))$ for any rule $r_j =< rb_j, re_j > (j \neq i))) \vee (Error(rb_i) \wedge (re_i = Permit) \wedge \neg rb_j$ for any permit rule $r_j =< rb_j, Permit > (j \neq i)) \vee (Error(rb_i) \wedge (re_i = Deny) \wedge \neg rb_j$ for any rule $r_j =< rb_j, re_j > (j \neq i))$.

When RCA =*Deny-unless-permit*, no permit rule before rule r_i should be fired to reach rule r_i. To make P and P' evaluate to different decisions, rule r_i in P should be fired. (1) If $re_i = Permit$, the propagation constraint requires that no permit rule after rule r_i should be fired. (2) If $re_i = Deny$, then P and P' have the same decision, regardless of the evaluation results of rules $r_{i+1}, ..., r_n$. To summarize, the missing rule must be the only permit rule that is fired in P. This is formalized by Theorem 10.

Theorem 10. Let $R = [r_1, ..., r_{i-1}, r_i, r_{i+1}, ..., r_n]$. $R' = [r_1, ..., r_{i-1}, r_{i+1}, ..., r_n]$. $rca(Deny\text{-}unless\text{-}permit, R, q) \neq rca(Deny\text{-}unless\text{-}permit, R', q)$ if and only if $re_i = Permit \wedge rb_i \wedge (\neg rb_j \vee Error(rb_j))$ for any permit rule $r_j =< rb_j, Permit > (j \neq i)$.

When RCA is *First-applicable*, all rules before rule r_i should evaluate to N/A, otherwise rule r_i in P will not be triggered. Note that if a rule before rule r_i evaluates to indeterminate, its decision, either ID or IP, becomes the decision of both P and P'. To make P and P' different, r_i in P should not evaluate to N/A, otherwise both P and P' depend on the evaluation results of rules $r_{i+1}, .., r_n$. In other words, r_i in P should fire or evaluate to indeterminate. In this case, its decision becomes P's decision. In P', however, rules $r_{i+1}, ..., r_n$ continue to be evaluated. To propagate the difference, the result of $rca(First\text{-}applicable, [r_{i+1}, \ldots r_n], q)$ must be different from r_i's decision. This is formalized by Theorem 11.

Theorem 11. Let $R = [r_1, ..., r_{i-1}, r_i, r_{i+1}, ..., r_n]$. $R' = [r_1, ..., r_{i-1}, r_{i+1}, ..., r_n]$. $rca(First\text{-}applicable, R, q) \neq rca(First\text{-}applicable, R', q)$ if and only if $\neg rb_j$ for any rule r_j $(0 < j < i) \wedge (rb_i \wedge re_i \neq rca(First\text{-}applicable, [r_{i+1}, ..., r_n], q) \vee Error(rb_i) \wedge I(re_i) \neq rca(First\text{-}applicable, [r_{i+1}, ..., r_n], q))$.

5 QUALITATIVE EVALUATION OF TESTING METHODS

This section applies the formalized FDCs to the qualitative evaluation of five testing methods for XACML3.0, including rule-coverage (RC), decision coverage (DC), no-error decision coverage (NE-DC), MC/DC, and no-error MC/DC (NE-MC/DC) [18]. They are the main methods publicly available for the current standard of XACML. The empirical studies showed that they have outperformed the previous methods. As they have all considered the reachability of rules and decision expressions (policy target, rule target, and rule condition), our discussion focuses on whether or not they satisfy the necessity and propagation constraints of each fault type. As mentioned before, other related work was based on the earlier versions of XACML, and thus is difficult to compare. Nevertheless, it appears that the rule coverage-based test selection [4] is comparable to RC, and XPTester [12] comparable to NE-DC.

Tables 6 and 7 summarize the analysis results. Table 6 presents the fault detection capabilities of the testing methods with respect to the incorrect use of rule-combining algorithms (i.e., mutants created by the CRC operator). For each pair of rule-combining algorithms $< RCA_1, RCA_2 >$, the mutation is either changing RCA_1 to RCA_2 or RCA_2 to RCA_1. Table 7 shows their fault detection capabilities with respect to other fault types. In each table, "Yes" (or "No") means that the mutants created by the corresponding mutation operator are "always" (or "never") killed. "Yes*" means that the mutants are conditionally killed. The conditions are mostly about propagation constraint and can further be refined according to structural patterns of policies (e.g., permit-only or deny-only rules). "No*" means that, generally, the testing method cannot kill the mutants, but exceptions may exist. For either "Yes*" or "No*", some mutants are killed, while others are live. However, we use "No*" to indicate that the cases of killed mutants are exceptional or coincidental. Table 7 covers all the rule-combining algorithms, where each entry indicates the minimum capability. For example, when the rule-combining algorithm is *First-applicable*, all entries of the CRE row is unconditional "Yes", because the decision of a fired rule becomes the policy decision. An unconditional "Yes" or "No" applies to any rule-combining algorithm, whereas a conditional "Yes*" or "No" may imply different conditions for different rule-combining algorithms. FPR and FDR apply only when the rule-combining algorithm is First-applicable.

The error tests, which are the difference between NE-DC and DC and between NE-MC/DC and MC/DC, can kill more mutants of RTT, RTF, RCT, RCF, ANF, and RNF. Compared to DC, MC/DC deals with the necessity constraint of most all RPTE mutants of compound policy and rule targets. MC/DC may also kill additional mutants of CRE, RTT, RTF, RCT, RCF, ANF, BNF, and RER. Nevertheless, MC/DC does not directly satisfy their propagation constraints, which largely depend on concrete policies. Therefore, in Table 7, the entries of MC/DC are mostly conditional "Yes*", rather than unconditional "Yes".

In the following, we elaborate on how the qualitative evaluation was performed using *Permit-overrides* as the rule-combining algorithm. Although the evaluation is independent of specific policies, we use the *Sample-PO* policy for illustration purposes. Table 8 presents all the test cases generated by the five testing methods,

Table 6: Fault detection capabilities of incorrect rule-combining algorithms (PO: Permit-overrides, DO: Deny-overrides, FA: First-applicable, DUP: Deny-unless-permit, PUD: Permit-unless-deny)

CRC: Change RCA	RC	NE-DC	DC	NE-MC/DC	MC/DC
PO vs DO	No*	Yes	Yes	Yes	Yes
PO vs DUP	No	Yes	Yes	Yes	Yes
PO vs PUD	No*	Yes	Yes	Yes	Yes
PO vs FA	No*	No*	Yes	No*	Yes
DO vs DUP	No*	Yes	Yes	Yes	Yes
DO vs PUD	No	Yes	Yes	Yes	Yes
DO vs FA	No*	No*	Yes	No*	Yes
DUP vs PUD	No*	Yes	Yes	Yes	Yes
DUP vs FA	No*	Yes	Yes	Yes	Yes
PUD vs FA	No*	Yes	Yes	Yes	Yes

Table 7: Fault detection capabilities of other fault types

Mutation operator	RC	NE-DC	DC	NE-MC/DC	MC/DC
PTT	No	Yes	Yes	Yes	Yes
PTF	Yes	Yes	Yes	Yes	Yes
CRE	Yes*	Yes*	Yes*	Yes*	Yes*
RTT	No*	Yes*	Yes*	Yes*	Yes*
RTF	Yes*	Yes*	Yes*	Yes*	Yes*
RCT	No*	Yes*	Yes*	Yes*	Yes*
RCF	Yes*	Yes*	Yes*	Yes*	Yes*
ANF	Yes*	Yes*	Yes*	Yes*	Yes*
RNF	Yes*	Yes*	Yes*	Yes*	Yes*
FPR	No*	No*	No*	Yes*	Yes*
FDR	No*	No*	No*	Yes*	Yes*
RER	Yes*	Yes*	Yes*	Yes*	Yes*
RPTE	No*	No*	No*	Yes*	Yes*

where (wt) denotes that the value has a wrong type for the attribute. Table 9 shows whether or not each of the sample mutants in Section III is killed by each testing method. If killed, the test cases are listed.

5.1 Rule Coverage

The RC method generates exactly one test to fire each rule. The test generation constraint for rule r_i is the conjunction of the reachability constraint of r_i and $rt_i \wedge rc_i$. The test that covers rule r_i also satisfies PT and $\neg rt_j \vee \neg rc_j$ for any permit rule $r_j =< rt_j, rc_j, Permit >$ before r_i. Tests 1-6 in Table 8 form the RC test suite for *Sample-PO*. Although each RC test fires a rule, the rule's effect does not necessarily become the policy's decision. For example, if the two deny rules were listed before the permit rules in *Sample-PO*, the tests covering the deny rules would lead to a policy decision of IP. Even though a deny rule is fired, the permit rules continue to be evaluated. These tests would make the permit rules evaluate to IP because there is no value for the title attribute.

Table 8: Sample-PO Test Cases Created by Testing Methods RC(1), NE-DC(2), DC(3), NE-MC/DC(4), MC/DC(5)

#	Test Input: < *department, title, job-class, location*>	Oracle value	Test Method 1	2	3	4	5
1	HR, director, ,on-campus	Permit	x	x	x	x	x
2	IT, director, ,off-campus	Permit	x	x	x	x	x
3	HR, deputy, ,on-campus	Permit	x	x	x	x	x
4	HR, deputy, ,off-campus	Permit	x	x	x	x	x
5	HR, ar, guest, off-campus	Deny	x	x	x	x	x
6	HR, ar, intern, off-campus	Deny	x	x	x	x	x
7	A, , ,	N/A		x	x	x	x
8	HR, aaI, , aas	ID		x	x	x	x
9	HR, director, , aaaaaan	ID		x	x	x	x
10	HR, deputy, , av	ID		x	x	x	x
11	HR, deputy, aC, k	N/A		x	x	x	x
12	IT, director, guest, aav	N/A		x	x	x	x
13	IT, director, intern, aav	N/A		x	x	x	x
14	A(wt), , ,	N/A			x		x
15	HR, aaaI(wt), , aas	IDP			x		x
16	HR,director, ,on-campus(wt)	IDP			x		x
17	HR,deputy, ,on-campus(wt)	IDP			x		x
18	HR,ar,guest,off-campus(wt)	N/A			x		x
19	HR,ar,intern,off-campus(wt)	N/A			x		x
20	HR,ar,part-time,off-campus	Deny				x	x
21	HR,ar,contractor,off-campus	Deny				x	x

Table 9: Testing results of the sample mutants (Y: Yes, N: No)

Mutant name	RC	NE-DC	DC	NE-MC/DC	MC/DC
PTT	N	Y(7)	Y(7,14)	Y(7)	Y(7,14)
RPTE0-1	N	N	N	Y(2)	Y(2)
CRE3	Y(3)	Y(3)	Y(3)	Y(3)	Y(3)
RTT5	N	N	N	N	N
RPTE5-1	N	N	N	Y(20)	Y(20)
RCT3	N	Y(10,11)	Y(10,11,17)	Y(10,11)	Y(10,11,17)
CRC-FA	N	N	Y(15-19)	N	Y(15-19)

5.1.1 CRC (Change Rule-Combining algorithm). RC may or may not kill the CRC/*Deny-overrides* mutant (i.e., *Permit-overrides* is changed to *Deny-overrides*). When *Permit-overrides* and *Deny-overrides* are non-equivalent, P contains a pair of permit and deny rules that conflict with each other. Suppose $r_i =< rb_i, re_i >$ appears before $r_j =< rb_j, re_j > (i < j)$, rb_i and rb_j are not identical, and $re_i \neq re_j$. The test that fires rule r_i does not kill the mutant unless it happens to (a) fire rule r_j or (b) evaluate r_j to indeterminate. (a) is unlikely because rb_j is not used for solving rb_i. The test that fires rule r_j does not fire rule r_i due to r_j's reachability constraint. (b) can be satisfied only when rules r_i and r_j involve different attributes and the test covering rule r_i has no value for some attribute in rule r_j. The CRC/*Deny-overrides* mutant of *Sample-PO* falls into the case of (b) and is killed by the RC method.

RC cannot kill the CRC/*Deny-unless-permit* mutant. When a RC test makes P evaluate to a permit decision, a permit rule was fired. So P' also results in a *Permit* decision. When a RC test makes P

evaluate to a *deny* decision, a deny rule was fired but no permit rule was fired. P' also results in a *Deny* decision. Here non-equivalence of *Permit-overrides* and *Deny-unless-permit* (or *Permit-unless-deny*) does not require that P should contain both permit and deny rules.

RC cannot kill the CRC/*Permit-unless-deny* mutant where *Permit-overrides* is changed to *Permit-unless-deny* except that a test covering a permit (or deny) rule also fires a deny (or permit) rule. In the exceptional case, the decision of P is $Permit$, whereas the decision of P' is *Deny*. This is unlikely because RC aims at covering individual rules.

RC cannot kill the CRC/*First-applicable* mutant where *Permit-overrides* is changed to *First-applicable* except that the test covering a deny rule also fires a permit rule after it (the decision of P is $Permit$, whereas the decision of P' is *Deny*). As shown in Table 9, RC cannot kill *Sample-PO-CRC-FA*.

5.1.2 PTT (Policy Target True). The RC method cannot kill PTT mutants because every RC test makes the policy target true. The necessity constraint is not satisfied. As shown in Table 9, for example, the RC test suite cannot kill *Sample-PO-PTT*. RC kills all PTF (policy target false) mutants because they result in an N/A policy decision for each test.

5.1.3 CRE (Change Rule Effect). RC can kill CRE mutants with possible exceptions. The test covering the mutated rule r_i meets the necessity constraint. It kills the mutant as along as it fires no permit rule after r_i so that the propagation constraint is satisfied. In Table 9, test 3 kills *Sample-PO-CRE3*. It satisfies the fault detection condition *department* $\in$ *{"HR", "IT"}* $\wedge$ *(title="deputy")* $\wedge$ *(location="on-campus")* described in Section IV.

5.1.4 RTT (Rule Target True). RC cannot kill RTT mutants with possible exceptions. For an RTT mutant with $r_i =< true, rc_i, re_i >$, the test covering the mutated rule $r_i =< rt_i, rc_i, re_i >$ does not kill it because the necessity constraint is not satisfied. An exception is that there exists another rule $r_j =< rt_j, rc_j, re_j >$ such that rt_j implies $\neg rt_j$ (e.g., *title="director"* implies *title* $\neq$ *"deputy")* and rc_j implies rc_i (e.g., they are equal or rc_i is empty). The test covering r_j makes r_i not-applicable in P but fired in P'. To propagate the difference, r_j should be a deny rule, otherwise both P and P' result in a $Permit$ decision. This implies that r_i should be a permit rule otherwise both P and P' evaluate to deny once r_j is fired. In addition, the test should not fire any other permit rule. To summarize, the PTT mutant is killed only when the mutated rule r_i is a permit rule and there exists a deny rule $r_j =< rt_j, rc_j, Deny >$ such that $rt_j \wedge rc_j \wedge \neg rt_i \wedge rc_i \wedge re_i = Permit \wedge (\neg rb_k \vee Error(rb_k))$ for any permit rule $r_k =< rb_k, Permit > (k \neq j, k \neq i)$. In Table 9, RC does not kill *Sample-PO-RTT5*. None of its tests satisfies the FDC *department* $\in$ *{"HR", "IT"}* $\wedge$ *title* $\notin$ *{"deputy", "director"}* $\wedge$ *job-class* $\notin$ *{"guest", "part-time", "intern", "contractor"}* $\wedge$ *(location="off-campus")* described in Section IV.

5.1.5 RCT (Rule Condition True). Similarly, RC cannot kill RCT mutants although there exist exceptions. For an RCT mutant $r_i =< rt_i, true, re_i >$, the test covering the mutated rule $r_i =< rt_i, rc_i, re_i >$ in P does not kill it because the necessity constraint is not satisfied. The exception is that there exists another rule $r_j =< rt_j, rc_j, re_j >$ such that rt_j implies rt_j (e.g., they are equal or rt_i is empty) and rc_j implies $\neg rc_i$ (e.g., *location="on-campus"* implies *location* $\neq$ *"off-campus"*). In this case, the test covering r_i makes r_i not-applicable in P but fired in P'. Similar to the above discussion on RTT, the RCT mutant is killed when r_i is a permit rule and there exists a deny rule $r_j =< rt_j, rc_j, Deny >$ such that $rt_j \wedge rc_j \wedge rt_i \wedge \neg rc_i \wedge re_i = Permit \wedge (\neg rb_k \wedge Error(rb_k))$ for any permit rule $r_k =< rb_k, Permit > (k \neq j, k \neq i)$. In Table 9, RC does not kill *Sample-PO-RCT3*.

5.1.6 RTF (Rule Target False), RCF (Rule Condition False), ANF (Add Negation Function), and RNF (Remove Negation Function). For each RTF (rule target false) mutant, the test that covers the mutated rule r_i has satisfied the reachability and necessity constraints. Generally, RC can kill an RTF mutant if and only if this test does not fire any permit rule after r_i (or any other rule) when r_i is a permit (or deny) rule. This is similar for RCF (rule condition false), ANF (add negation function), and RNF (remove negation function).

5.1.7 RER (REmove a Rule). RC may or may not kill RER mutants. For an RER mutant where rule $r_i =< rb_i, re_i >$ is missing, the test covering r_i evaluates r_i to re_i. If $r_i = Permit$, then the mutant is killed if the test fires no other permit rule. Consider rules $r_1 =< a \vee b, Permit >$ and $r_2 =< a \vee c, Permit >$ that are overlapping, but not subsumed by each other. The RC test for r_1 (or r_2) that satisfies a does not kill the RER mutant where r_1 (or r_2) is removed. If $r_i = Deny$, the mutant is killed when no other rule is fired and there does not exist a pair of permit and deny rules that both evaluate to indeterminate.

5.1.8 RPTE (Remove Parallel Target Element). RC cannot kill RPTE mutants with possible exceptions. As shown in Table 10, RC does not kill *Sample-PO-RPTE0-1* or *Sample-PO-RPTE5-1*. RPTE applies to a policy or rule target of the form $c_1 \wedge ... \wedge c_n$ or $c_1 \vee ... \vee c_n$ $(n > 1)$. A test that satisfies $c_1 \wedge ... \wedge c_{i-1} \wedge c_i \wedge c_{i+1} ... \wedge c_n$ also satisfies $c_1 \wedge ... \wedge c_{i-1} \wedge c_{i+1} ... \wedge c_n$ where c_i is removed. A test that satisfies $c_1 \vee ... \vee c_{i-1} \vee c_i \vee c_{i+1} ... \vee c_n$ also satisfies $c_1 \vee ... \vee c_{i-1} \vee c_{i+1} ... \vee c_n$ with c_i removed except that c_i is true and each c_j $(j \neq i)$ is false. So the necessity constraint is not met. RC may kill $\frac{1}{n}$ of the RPTE mutants where n is the number of parallel elements in $c_1 \vee ... \vee c_{i-1} \vee c_i \vee c_{i+1} ... \vee c_n$. In *Sample-PO*, $n = 2$ for each compound expression. RC kills *Sample-PO-RPTE5-0* where the first parallel target element *job-class="guest"* is removed. The test that covers rule R5 evaluates the mutated target of rule R5 to false.

5.2 Decision Coverage and Non-Error Decision Coverage

The DC method derives tests to make the policy target evaluate to $true$, $false$, and $Error$, respectively, and the rule target (condition) of each rule evaluate to $true$, $false$, and $Error$, respectively. The "true" case of one-level decision expression is used as a reachability constraint for the next level decision expression (e.g., policy target vs rule target, and rule target vs rule condition). NE-DC is a special case of DC without error tests. It ensures that, for each rule $r_i =< rt_i, rc_i, re_i >$, there are three tests to cover the following combinations of rule target and rule condition: $rt_i \wedge rc_i$, $rt_i \wedge \neg rc_i$, and $\neg rt_i$. When solving $\neg rt_i$, it also tries to satisfy rc_i if feasible. In other words, $\neg rt_i$ is likely $\neg rt_i \wedge rc_i$ (including the case of rc_i being empty). $rt_i \wedge rc_i$ is also in the RC test suite. $\neg rt_i$ (or $rt_i \wedge \neg rc_i$

when rt_i is empty) might be covered by the RC test suite if the rule is a permit rule, but not the last rule in the policy. The reachability constraint of the rules after it requires that either its target or its condition should evaluate to false. Generally, NE-DC has a test that makes the last rule not applicable and thus leads the policy to an N/A decision. As shown in Table Table 8, the NE-DC test suite for *Sample-PO* has 13 tests, including the six RC tests. DC adds six more error tests to NE-DC. Note that, when NE-DC and DC test suites are generated separately, the random values created by the constraint solver could be different. An example is *"A"* for the attribute *department* in test 7. For convenience, Table 8 uses the same random value for all test suites so that they can fit in one table to demonstrate the relationships among the testing methods. This does not affect fault detection. This is similar for NE-MC/DC and MC/DC.

NE-DC is similar to RC for the following mutants: PTF, CRC/*Deny-overrides*, CRC/*First-applicable*, CRE, RTF, RCF, ANF, BNF, and RER.

5.2.1 CRC (Change Rule-Combining algorithm). NE-DC kills the CRC/*Deny-unless-permit* mutant where *Permit-overrides* is changed to *Deny-unless-permit*. Consider the last rule $r_n =< rt_n, rc_n, re_n >$. There is a test for $\neg rt_n$ or $rt_n \wedge \neg rc_n$, which evaluates the rule to N/A. In this case, no permit rule is fired. Assuming that this test fires no deny rule, the decision of P is not deny, whereas the decision of P' is deny.

NE-DC kills the CRC/*Permit-unless-deny* mutant where *Permit-overrides* is changed to *Permit-unless-deny*. Consider the last rule $r_n =< rt_n, rc_n, re_n >$. There is a test for $\neg rt_n$ or $rt_n \wedge \neg rc_n$, which evaluates the rule to N/A. In this case, no permit rule is fired. The decision of P is not *Permit*, whereas the decision of P' is *Permit* assuming that this test fires no deny rule.

5.2.2 PTT (Policy Target True). NE-DC kills PTT mutants because it has a test that evaluates the policy target to false. The decision of P is N/A, whereas a PTT mutant typically evaluates to indeterminate because of missing attribute values when the rules are evaluated. For *Sample-PO*, test 7 makes the policy target false and the policy evaluate to N/A. It kills *Sample-PO-PTT*, which results in *IDP* because each permit/deny rule evaluates to IP/ID and the combined decision is *IDP*.

5.2.3 RTT (Rule Target True). NE-DC kills RTT mutants although there are exceptions. Consider an RTT mutant with $r_i =< true, rc_i, re_i >$. The test for $\neg rt_i$ likely satisfies the necessity constraint $\neg rt_i \wedge rc_i$. As mentioned before, when solving $\neg rt_i$, NE-DC considers rc_i if feasible. This test makes r_i not-applicable in P but fired in P'. The necessity constraint can also be satisfied by a NE/DC test for a different rule $r_j =< rt_j, rc_j, re_j >$ such that rt_j implies $\neg rt_j$ and $\neg rc_j$ implies rc_i. This test for $rt_j \wedge \neg rc_j$ makes r_i not-applicable in P but fired in P'. The effect of rule r_j does not matter. Either of the above tests kills the mutant if (a) it does not fire any other permit rule, and (b) it fires no other deny rule and makes no permit rule evaluate to indeterminate if $re_i = Deny$.

5.2.4 RCT (Rule Condition True). NE-DC can kill RCT mutants with exceptions. Consider an RCT mutant with $r_i =< rt_i, true, re_i >$. The new test for $rt_i \wedge \neg rc_i$ always satisfies the necessity constraint. It makes r_i not-applicable in P and but fired in P'. It kills the RCT mutant if it does not fire any other rule, which is very likely. As shown in Table 9, tests 10 and 11 of *Sample-PO* kill *Sample-PO-RCT3*.

5.2.5 RPTE (Remove Parallel Target Element). NE-DC cannot kill RPTE mutants with exceptions. NE-DC introduces a new test for $\neg(c1 \wedge ... \wedge c_n)$ or $\neg(c_1 \vee ... \vee c_n)$. A test that satisfies $\neg(c_1 \wedge ... \wedge c_{i-1} \wedge c_i \wedge c_{i+1} ... \wedge c_n)$ also satisfies $\neg(c_1 \wedge ... \wedge c_{i-1} \wedge c_{i+1} ... \wedge c_n)$ except that c_i is false and c_j is true for all $j \neq i$ (i.e., NE/DC may kill $\frac{1}{n}$ of the RPTE mutants). A test that satisfies $\neg(c_1 \vee ... \vee c_{i-1} \vee c_i \vee c_{i+1} ... \vee c_n)$ also satisfies $\neg(c_1 \vee ... \vee c_{i-1} \vee c_{i+1} ... \vee c_n)$. It does not kill additional RPTE mutants.

5.2.6 Summary. To summarize, in addition to the mutants killed by the RC method, NE-DC improves the fault detection capability by dealing with PTT, CRC/*Deny-unless-permit*, CRC/*Permit-unless-deny*, and RCT mutants.

Compared to NE-DC, DC introduces error tests for policy target, rule target and condition. Error tests do not kill any mutants of CRE, ANF, RNF, and RPTE (where the error does not occur in the removed clause) because the mutation point evaluates to the same type of indeterminate in both the original policy and the mutant. However, they satisfy the necessity constraints of mutants created by PTT, PTF, RTT, RTF, RCT, RCF, RER, and RPTE (where the error occurs only in the removed clause). They kill PTF mutants when the rule-combining result is not N/A, and PTF mutants when the rule-combining result is *Permit* or *Deny*. We will not elaborate on this as these mutants are already killed by other tests. For other mutants listed above, propagation of the difference requires that no other rule should evaluate to the same type of indeterminate. This indicates that other rules should not involve the same attribute that causes the error occurrence. It can be hard to satisfy this constraint because rules in the same policy are usually defined over the same set of attributes – error occurrence in one rule implies the occurrence of the same error in all other rules defined over the same attributes. As shown in the empirical study, there are only a small number of error tests. Error tests for rules can kill the CRC/*First-applicable* mutant (e.g., *Sample-PO-CRC-FA* in Table 9). This requires that an error occurs in both permit and deny rules so that the result of *Permit-overrides* is *IDP* whereas the result of *First-applicable* is *IP* or *ID*. This is the case when *Permit-overrides* and *First-applicable* are non-equivalent: there are both deny and permit rules defined over the same set of attributes. In addition, error tests can reveal incorrect attribute types, which are not handled by the existing mutation operators.

5.3 MC/DC and Non-Error MC/DC

The MC/DC method generates tests to achieve MC/DC and cover error occurrence for each decision expression. It is different from DC only when the decision expression has one or more logical connectives. For conjunction $c_1 \wedge ... \wedge c_n$, MC/DC produces $n + 1$ tests: one test that evaluates all c_i to true and n tests that each evaluates one c_i to false and all other c_j to true. For $c_1 \vee ... \vee c_n$, MC/DC creates $n + 1$ tests including one that evaluates all c_i to false and n tests that each evaluates one c_i to true and all other c_j to false. NE-MC/DC is a special case of MC/DC without error tests. For *Sample-PO*, MC/DC is different from DC because of the policy target, rule R5, and rule R6. As shown in Table 8, test 2

of MC/DC (or NE-MC/DC) uses *"IT"* for *department* to achieve a different coverage of the policy target. Tests 20 and 21 deal with rules R5 and R6, respectively.

The mutants killed by NE-DC are also killed by NE-MC/DC. If it is a "Yes*" for NE-DC, it remains unchanged for NE-MC/DC. However, NE-MC/DC may kill additional mutants of RTT, RTF, RCT, RCF, ANF, BNF, and RER because it creates more tests for compound rt_i and rc_i (e.g., $\neg rt_i$ and $rt_i \wedge \neg rc_i$ for killing RTT mutants discussed before). These tests increase the chance of meeting the propagation requirements related to the evaluations of other rules.

Compared to NE-DC, a major new capability of NE-MC/DC is the detection of faults in RPTE mutants. It always satisfies the necessity constraint. If $\omega = c_1 \wedge ... \wedge c_{i-1} \wedge c_i \wedge c_{i+1} ... \wedge c_n$ and $\omega' = c_1 \wedge ... \wedge c_{i-1} \wedge c_{i+1} ... \wedge c_n$ where c_i is removed. NE-MC/DC has a test where c_i is false and c_j is true for all $j \neq i$. This test makes ω false and ω' true. If $\omega = c_1 \vee ... \vee c_{i-1} \vee c_i \vee c_{i+1} ... \vee c_n$ and $\omega' = c_1 \vee ... \vee c_{i-1} \vee c_{i+1} ... \vee c_n$ where c_i is removed. NE-MC/DC has a test where c_i is true, and c_j is false for all $j \neq i$. It makes ω true and ω' false. When ω is the policy target, the propagation constraint is satisfied, assuming the rules use at least one attribute that does not appear in ω. In Table 9, test 2 of NE-MC/DC kills *Sample-PO-RPTE0-1* as it evaluates the mutated policy target to N/A.

Consider RPTE applied to the mutated rule $r_i =< rt_i, rc_i, re_i >$. When $rt_i = c_1 \wedge ... \wedge c_{i-1} \wedge c_i \wedge c_{i+1} ... \wedge c_n$ and the MC/DC test makes rt_i false and rt_i' true. It also makes rc_i true (or rc_i is empty) as the MC/DC method attempts to satisfy rc_i if feasible when solving $\neg rt_i$. When $rt_i = c_1 \vee ... \vee c_{i-1} \vee c_i \vee c_{i+1} ... \vee c_n$, the MC/DC test for rt_i being true and rt_i' being false makes rc_i true because it is used as part of the reachability constraint of rc_i. So, the test kills the mutant if (a) it does not fire any other permit rule, and (b) it fires no other deny rule and makes no permit rule evaluate to indeterminate if $re_i = Deny$. For *Sample-PO*, test 20 kills *Sample-PO-RPTE5-1* as shown in Table 9.

6 CONCLUSIONS

We have presented the sufficient and necessary conditions for detecting XACML policy faults according to a comprehensive fault model. The conditions allow the fault detection capabilities of testing methods to be qualitatively evaluated by identifying whether or not they can detect each type of faults. This is distinct from the existing research that solely relies on mutation experiments with sample policies.

In this paper, the fault detection conditions are formalized with respect to fault types, not limited to the existing mutation operators. They can be instantiated and simplified whenever a new mutation operator is introduced for a fault type. The formalized fault detection conditions also provide a fundamental guideline for the design of effective testing methods for access control policies in XACML and other similar languages. Although the existing work has commonly applied mutation experiments to the quantitative measurement of testing effectiveness, the fault model used to derive mutants is essentially an afterthought, not a built-in component of testing methods. Because the main purpose of testing is to find potential faults in a given policy, understanding of fault detection conditions is key to effective testing. An effective method should target specific types of policy faults and deal with the reachability, necessity, and propagation constraints of fault detection.

Our future work aims to investigate fault detection conditions of NGAC (Next Generation Access Control) [9] policies based on the fault model and mutation operators [5]. Although NGAC and XACML are both ABAC standards, they are very different. XACML specifies policies through logical rules defined over attributes, whereas NGAC expresses policies through configurations of relations among attributes (e.g., assignments, associations, prohibitions, and obligations). As the reference implementation of NGAC has become available, testing methods have begun to emerge [5].

ACKNOWLEDGMENTS

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

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 Conference on Web Information Systems and Technologies (WEBIST)*. 155–160.
[3] 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)*.
[4] 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 Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 12–21.
[5] E. Chen, V. Dubrovenski, and D. Xu. 2021. Mutation Analysis of NGAC Policies. In *Proc. of the 26th ACM Symposium on Access Control Models and Technologies (SACMAT'21)*.
[6] 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.
[7] R.A. DeMillo and A. J. Offut. 1991. Constraint-based automatic test data generation. *IEEE Trans. on Software Engineering* 17, 9 (9 1991), 900–910.
[8] 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.
[9] W. Jansen (Ed.). 2020. *Next Generation Access Control (NGAC), DPANS INCITS 565, Revision 1.00.*
[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] 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.
[12] 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)*.
[13] E. Martin and T. Xie. 2006. Automated test generation for access control policies. In *Supplemental Proc. of ISSRE*.
[14] 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.
[15] 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.
[16] OASIS. 2013. *eXtensible Access Control Markup Language (XACML) Version 3.0.* http://www.oasisopen.org/committees/xacml/.
[17] 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.
[18] 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.
[19] D. Xu, R. Shrestha, and N Shen. 2020. Automated strong mutation testing of XACML policies. In *Proc. of the 25th ACM Symposium on Access Control Models and Technologies (SACMAT'20)*. 105–116.