STRONG MUTATION-BASED TEST GENERATION OF XACML POLICIES

by

Roshan Shrestha

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

December 2018

© 2018

Roshan Shrestha

ALL RIGHTS RESERVED

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Roshan Shresthat

Thesis Title: Strong Mutation-Based Test Generation of XACML Policies

Date of Final Oral Examination: 24 October 2018

The following individuals read and discussed the thesis submitted by student Roshan Shrestha, and they evaluated the student's presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Dianxiang Xu, Ph.D. Chair, Supervisory Committee

Edoardo Serra, Ph.D. Member, Supervisory Committee

Yantian Hou, Ph.D. Member, Supervisory Committee

The final reading approval of the thesis was granted by Dianxiang Xu, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

ACKNOWLEDGMENTS

I would like to thank Boise State University for giving me the opportunity to pursue graduate study without which I would not have come this far. I would like to express the highest gratitude to the Computer Science Department of Boise State University for always supporting me to achieve academic excellence.

I am grateful and have great appreciation to Dr. Dianxiang Xu for including me in his research group. I highly admire the learning opportunity and environment he provided to me. His ideas and suggestions were crucial for me to achieve my goals. His achievements, his dedication and his passion for the research in Computer Security will always be a continuous source of inspiration for the rest of my life.

I am also very grateful to Dr. Edoardo Serra and Dr. Yantian Hou for being on my thesis committee and providing valuable feedback on my work.

Finally, I am grateful to my parents and my sister for always supporting me and encouraging me to achieve my goals.

ABSTRACT

There exist various testing methods for XACML policies which vary in their overall fault detection ability and none of them can detect all the (killable) injected faults except for the simple policies. Further, it is unclear that what is essential for the fault detection of XACML policies. To address these issues, we formalized the fault detection conditions in the well-studied fault model of XACML policies so that it becomes clear what is essential for the fault detection. We formalized fault detection conditions in the form of reachability, necessity and propagation constraint. We, then, exploit these constraints to generate a mutation-based test suite with the goal to achieve perfect mutation score. Additionally, we have empirically evaluated the cost-effectiveness of various coverage-based testing methods against the near-optimal test suite from strong mutation-based test generation (SMT). Rule coverage has good cost-effectiveness such that it achieved better MKPT scores than SMT in many of the policies; however, it has poor fault detection capability. Decision coverage is nearly as cost-effective as SMT in most of the policies and it achieves better mutation score than rule coverage but could not achieve good mutation score in many of the policies. MC/DC have slightly less MKPT scores than SMT; nonetheless, among coverage-based testing methods, MC/DC tests have the highest mutation score and hence could reveal most of the faults. MC/DC even achieved a perfect mutation score for some policies; however, it still could not maintain good mutation score in all the policies.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iv

ABSTRACT v

TABLE OF CONTENTS vi

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ABBREVIATIONS xi

CHAPTER 1 1

Introduction 1

1.1 Background 1

1.2 Thesis Statement 7

1.3 Method 9

1.4 Outline 14

CHAPTER 2 15

Background and Related Work 15

2.1 Mutation analysis and mutation-based test generation 15

2.2 Introduction to XACML 17

CHAPTER 3 31

Fault Detection Condition 31

3.1 FDC for Change Rule Effect (CRE) 35

3.2 FDC for Rule Target True (RTT)..48

3.3 FDC for Rule Target False (RTF) 59

3.4 FDC for Rule Condition True (RCT)..65

3.5 FDC for Rule Condition False (RCF) 72

3.6 FDC for Add Not Function (ANF)...78

3.7 FDC for Remove Not Function (RNF) 82

3.8 FDC for Remove a Rule (RER) 83

3.9 FDC for Policy Target True (PTT) 89

3.10 FDC for Policy Target False (PTF)..91

3.11 FDC for First Permit Rule (FPR) 93

3.12 FDC for First Deny Rule (FDR) 94

3.13 FDC for Remove Parallel Target Element (RPTE)........................94

3.14 FDC for Change Rule Combining Algorithm (CRC) 100

CHAPTER 4 109

Mutation-Based Test Generation 109

4.1 Strong Mutation-Based Test Generation..110

4.2 Return Constraint 117

4.3 FDC Constraint for Propagation 117

4.4 FDC Constraint and Tests Generation for RTT 121

4.5 FDC constraint and tests generation for RTF................................122

4.6 FDC Constraint and Test Generation for RCT...............................123

4.7 FDC Constraint and Test Generation for RCF...............................124

4.8 FDC Constraint and Test Generation for ANF 124

4.9 FDC Constraint and Test Generation for RNF 124

4.10 FDC Constraint and Test Generation for RER 124

4.11 FDC Constraint and Test Generation for PTT 125

4.12 FDC Constraint and Test Generation for PTF 125

4.13 FDC Constraint and Test Generation for FPR 125

4.14 FDC constraint and test generation for FDR 126

4.15 FDC Constraint and Test Generation for RPTE 127

4.16 FDC constraint and test generation for CRC 129

CHAPTER 5 130

Quantitative Analysis 130

5.1 Experiment Setup 130

5.2 Results 133

5.3 Threats to Validity 150

CHAPTER 6 152

Conclusions 152

6.1 Summary 152

6.2 Future work 153

REFERENCES 155

LIST OF TABLES

Table 3.1: Fault Model 31

Table 3.2: A Faulty Policy with an Incorrect Rule Effect 36

Table 3.3: Possible Evaluations Of Other Rules when Rb_i Evaluates to True 39

Table 3.4: Possible Evaluations of Other Rules when Rb_i Evaluates to Error 41

Table 3.5: Possible Evaluations of Other Rules When Rt_i Evaluates to *N/A* and R_i Is *Deny* Rule in P 53

Table 3.6. Possible Evaluations of Other Rules When Rt_i Evaluates to *N/A* And R_i is *Permit* Rule in P 54

Table 3.7: Possible Evaluations of Other Rules When Rt_i Evaluates to *Error* And R_i is *Deny* Rule in *P* 55

Table 3.8. Possible Evaluations of Other Rules when Rt_i Evaluates to Error and R_i is *Permit* Rule in P 56

Table 5.1: Policies used for the experiment 132

Table 5.2: Number of tests generated 135

Table 5.3: Test generation time (in milliseconds) 138

Table 5.4: Mutation Scores 141

Table 5.5: Live Mutants 144

Table 5.6: MKPT Scores 148

LIST OF FIGURES

Figure 1.1: ABAC Concept [1] 3

Figure 1.2: Data-flow in ABAC [2] 5

Figure 2.1: XACML policy language model [2] 18

Figure 2.2. A sample XACML policy [13] 22

LIST OF ABBREVIATIONS

XACML - eXtensible Access Control Markup Language

AC - Access Control

ABAC - Attribute-based access control

RBAC - Role-based access control

RCA - Rule Combining Algorithm

PCA - Policy Combining Algorithm

MKPT - Mutants killed per unit test

PT - Policy Target

PST - Policy set Target

PO - Permit-overrides

DO - Deny-overrides

PUD - *permit-unless-deny*

DUP - *deny-unless-permit*

FA - *first-applicable*

OPO - Ordered *permit-overrides*

ODO - Ordered *deny-overrides*

RC - Rule Coverage

NE-DC - Non-error Decision Coverage

DC - Decision Coverage

MC/DC - Modified Condition/Decision Coverage

NE-MC/DC - Nor-*error* Modified Condition/Decision Coverage

FDC - Fault detection condition

PEP - Policy Enforcement Point

PDP - Policy Decision Point

PAP - Policy Administratio Point

PIP - Policy Information Point

I(D) - Indeterminate Deny

I(P) - Indeterminate Permit

I(DP) - Indeterminate Deny/Permit

SMT - Strong mutation test generation

NO-SMT - Non-optimized Strong mutation test generation

CRE - Change Rule Effect

RTT - Rule Target True

RTF - Rule Target False

RCT - Rule Condition True

RCF - Rule Condition False

ANF - Add Not Function

RNF - Remove Not Function

RER - Remove a rule

PTT - Policy Target True

PTF - Policy Target False

FPR - First Permit Rule

FDR - First Deny Rule

RPTE - Remove Parallel Target Element

CRC - Change Rule Combining Algorithm

CHAPTER 1

Introduction

1.1 Background

Access Control (AC) mechanism is a fundamental security mechanism that serves to limit the access to a system with virtual or physical resources (objects) based on subject requesting the access. In AC models like IBAC (Identity Based Access Control) or RBAC (Role-Based Access Control), the solution is based primarily on the identity of a subject where access to an object will be individually granted to a locally identified subject or locally defined roles that the subject is a member of. The subject qualifiers, such as identity and roles, are often insufficient in the expression of real-world AC needs because real-world AC requirements often need to deal with environmental conditions. However, traditional AC models like RBAC cannot incorporate factors pertaining to environmental conditions effectively. Environmental conditions include operational or situational context which are detectable environmental characteristics in which an access request is made. Environmental characteristics may include the current time, day of the week, a location of a user, or the current threat level which are independent of subject or object [1].

Attribute-Based Access Control (ABAC) has emerged as a new generation of access control methods for tackling the afore-mentioned issues. ABAC avoids the need for capabilities (operation/object pairs) to be directly assigned to a subject (requesters) or to their roles or groups before the request is made. The ABAC engine can make an authorization

decision when a subject requests access based on the assigned attributes of the requester, the assigned attributes of the object, environmental conditions, and a set of policies that are specified in terms of those attributes and conditions. This arrangement enables policies to be created and managed without direct reference to potentially numerous users and objects. Further, users and objects can be provisioned without reference to a policy [1].

ABAC allows us to specify fine-grained access control by combining various attributes of authorization elements into access control decisions. The attributes are predefined characteristics of subjects (e.g., job title and age), resources (e.g., data, programs, and networks), actions, and environments (e.g., current time and IP address). In ABAC, the subject presents a request to access the resource (object) [1]. The ABAC mechanism, then, evaluates policies, subject attributes, object attributes and environmental conditions to make the access control decision for the request. If authorized, the subject is given access to the resource. This is the core concept of ABAC which is illustrated in Figure 1.1.

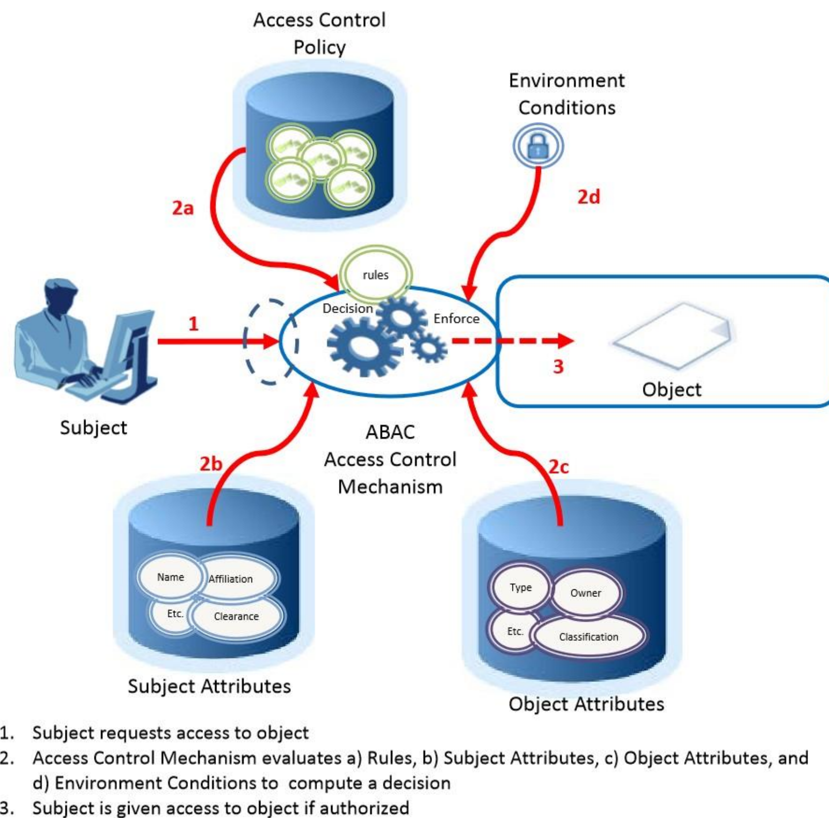

Figure 1.1: ABAC Concept [1]

XACML is an OASIS standard for ABAC policy specification. It provides both a policy language and an access control decision request/response language (both written in XML). The policy language is used to describe general access control requirements, and has standard extension points for defining new functions, data types, combining logic, etc. The request/response language is for defining a query to ask whether a given action should be allowed or not and interpreting the result. The response consists of a decision about whether the request should be permitted or not. The possible values in response are Permit, Deny, Indeterminate (an error occurred, or some required value was missing, so a decision cannot be made) or Non-applicable (the request can't be answered by this service) [2].

The typical setup in an ABAC system is that someone (subject) wants to take some action on a resource (object). A subject will make a request to the system which is intercepted by a

Policy Enforcement Point (PEP) which is responsible for protecting the resources (like a filesystem or a web server). Once PEP receives a request, it will send the request to a Policy Decision Point (PDP) which will determine policies that are applicable to the request and determines whether access should be granted or not. To make an access decision, PDP needs to evaluate certain attributes for which it makes a query to the Policy Information Point (PIP) which is responsible for resolving the attribute values required to make access decisions. Once the PDP gets the required attributes, it makes access decisions from the applicable policies. The response (decision) is then returned to the PEP which will enforce the access decision [2]. Figure 1.2 depicts the data flow in an ABAC system. It also consists of a Policy Administration Point (PAP) which is responsible for management and administration of policies themselves.

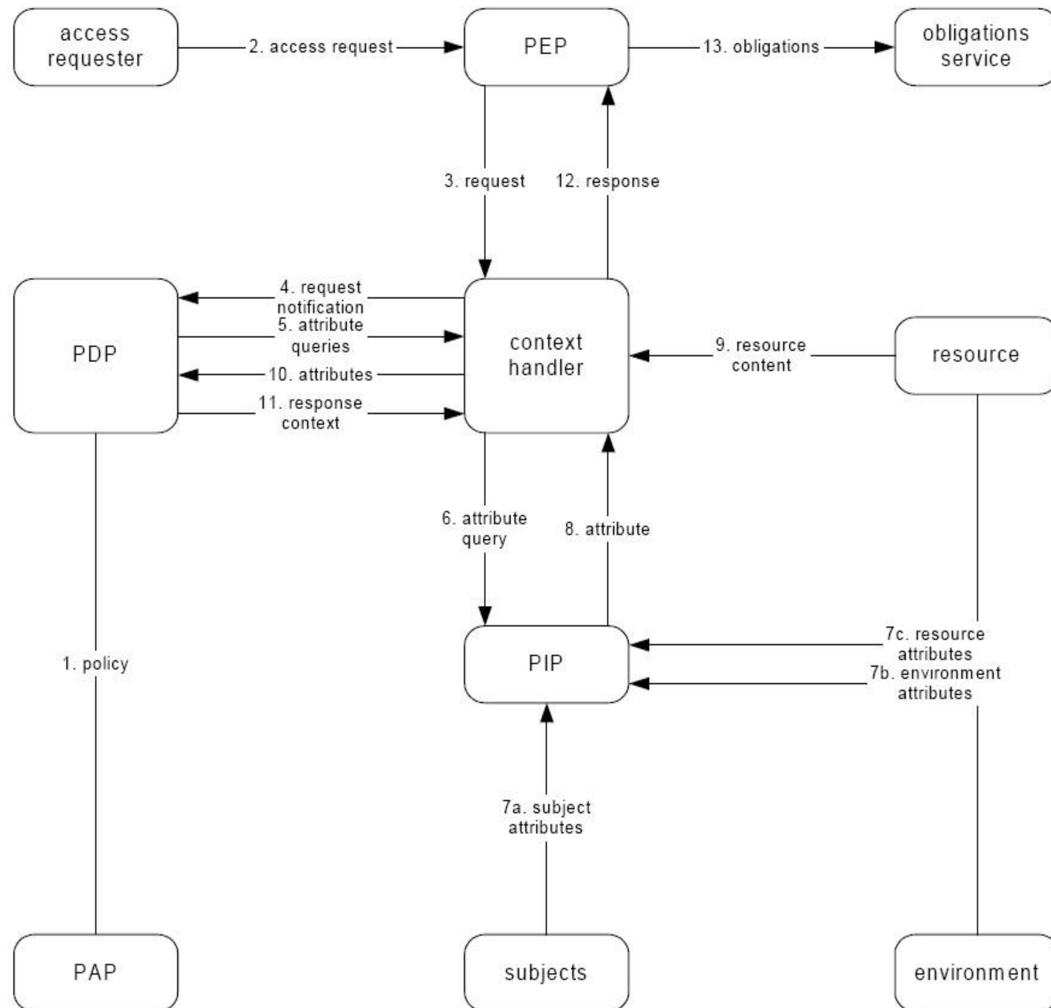

Figure 1.2: Data-flow in ABAC [2]

XACML policies may have various faults due to the misunderstanding of access control requirements, the complexity of access control language, and coding errors. While ABAC is more expressive than traditional AC methods such as RBAC, however, it is complex, and its complexity increases the likelihood of the existence of faults resulting in vulnerabilities as well as the level of difficulty in revealing these vulnerabilities. Research has shown that XACML policies are subject to a variety of faults, such as incorrect rule targets, incorrect rule conditions, incorrect rule effects, incorrect targets of policies and policy sets, and incorrect uses of the rules or policy combining algorithms [3, 4].

A major means for finding faults in XACML policy is to execute an access control system with a test suite (a set of test cases). A test case includes test input (access request) and corresponding test Oracle (expected response). A test fails when the system's actual response to the request is different from its expected response. Such failure often indicates the existence of a fault that may lead to unauthorized access or denial of service.

Current researches on testing XACML policies have commonly used policy mutation analysis to evaluate the fault detection ability of testing methods. It involves creating mutants of the policy under test using various mutation operators for a fault model (a set of the fault types). Each mutant is a variation of the given policy with an injected fault, which represents an error that a policy writer might make. A mutant is said to be killed if it fails one or more tests. The fault detection ability of the testing method is indicated by a mutation score which is the ratio between the number of mutants killed by the test suite and the total number of non-equivalent mutants.

Existing testing methods [5–12] vary in their overall fault detection ability, however, none of them can kill all the injected faults except for simple policies. The high-end ratios were usually obtained for simple policies and does not achieve such high-end ratios in complex policies. As a result, if these testing methods are applied in testing real-world access control systems with complex policy, it may not reveal many faults. As a result, it may leave significant vulnerabilities in the deployed system. In short, the existing approaches are inadequate for the high assurance of XACML-based access control. Moreover, it remains unclear what is essential to the fault detection given the fault and cost-effectiveness of testing various XACML testing methods.

To overcome afore-mentioned issues, we present a formalization of the fault detection conditions for illustrating what is essential for the fault detection of XACML policies in well-studied fault models [4,14,15] of XACML policies. The formalized fault detection condition is then used for formulating strong mutation-based test generation with the goal of achieving perfect mutation score. Finally, we have done the empirical study evaluating test suites from various coverage-based testing methods along with the test suites from strong mutation-based testing methods. The empirical study involves mutation analysis of various policies with different levels of complexity to identify fault detection capability and cost-effectiveness of each testing method.

1.2 Thesis Statement

The objective of this thesis is threefold. First, we formalize the fault detection conditions for faults in well-studied fault models of XACML policies. The goal is to make clear (formal) that what is essential for the fault detection given the fault. The fault detection condition of a given fault specifies the reachability, necessity, and propagation constraints that a test must satisfy in the order to reveal the fault. Although the notions of reachability, necessity, and propagation (a.k.a. sufficiency) constraints originate from mutation testing or constraint-based software testing [28,30], there is a lack of formal treatment of these constraints. In particular, the problem with propagation constraints of software is known to be intractable [29,30] because of the explosion of program execution paths. In this paper, the unique features of XACML make it feasible to formally represent the reachability, necessity, and propagation constraints of access control policies in three-valued logic.

Second, we formulate strong mutation-based test generation by exploiting the formalized fault detection condition to produce near-optimal test suites for XACML policies. A test suite for a given policy is said to be optimal if it contains the smallest number of tests which can achieve 100% mutation score. This paper considers near-optimality rather than strict optimality because constraints in XACML involve various data types, functions, and first-order predicates and solving the fault detection condition boils down to the constraint satisfaction problem, which is known to be undecidable. Nevertheless, due to the integration of all reachability, necessity, and propagation constraints for strong mutation testing, our approach is actually able to automatically generate the near-optimal test suites for all the XACML3.0 policies in the most recent literature [27]. This distinguishes our work from the existing mutation-based test generators that only deal with reachability and necessity constraints, a.k.a. weak mutation testing [20]. Generally, tests generated from weak mutation cannot achieve 100% mutation score [28, 31]. Further, SMT (strong mutation-based test generation) requires larger test generation time than all the testing methods discussed in this work. The objective of the SMT is to evaluate the cost-effectiveness and is not feasible for a policy with the larger number of the rules. Hence, we also presented NO-SMT (non-optimized strong mutation-based test generation) whose test generation time is greater than MC/DC but still comparable to it and is feasible to apply for larger policies.

Third, we present the quantitative evaluations of several test generation methods comparing against a near-optimal test suite (from SMT) of subject policies to establish cost-effectiveness and fault detection capability. The main testing methods for XACML

3.0 policies include rule coverage-based test generation, two forms of test generation with decision coverage, and two forms of MC/DC test generation [27].

1.3 Method

A fault in a policy is an *error* or flaw that causes it to produce an incorrect result. A fault may result in a different output than it is supposed to produce. We can exploit this difference in result between faulty policy and correct policy to reveal a fault. The idea is to record the output of the policy for some input when we know that policy is correct. Later, when we need to determine there exists fault or not, we supply those inputs to the suspect policy and if the result is different than the previously recorded result, we could conclude that there exists a fault. A fault policy, however, does not necessarily produce a different result than the correct policy for all possible inputs. The test inputs must satisfy certain constraint to produce a different result than from the correct policy and hence reveal the fault. Such constraints are referred to as fault detection conditions.

1.3.1 Fault Detection Condition (FDC)

The fault detection condition of a given fault specifies the constraints (or condition) that a test must satisfy to reveal the fault. A test input must satisfy the reachability, necessity and propagation condition to reveal the fault [18]. This is because to reveal the fault in a policy, the test must reach the faulty policy element which is termed as reachability condition. Once it is reached, the test must evaluate the faulty element to produce an incorrect intermediate result which is different than that from its correct counterpart which is referred to as necessity condition. If a test does not meet the reachability condition and/or necessity condition, the correct policy and faulty policy will behave the same and we do not have a means for detecting the fault. Once the incorrect intermediate result is produced,

it should play a role to produce a different result in the faulty policy than to the correct policy which is referred to as propagation condition. Hence, propagation condition is also essential for fault detection because reachability and necessity condition may only produce the incorrect intermediate result which may not be visible to the final access control decision if propagation condition is not met in which case we could not reveal the fault. Hence, we define fault detection condition with the following three constraints:

a) *Reachability(**R**) constraint*: the test must reach the faulty policy element (e.g., rule target, rule condition, rule effect, policy target, and combining algorithm).

b) *Necessity(**N**) constraint*: the test must make the faulty element evaluate to an incorrect intermediate result which is different from the evaluation result that should be produced by its correct counterpart.

c) *Propagation(**P**) constraint*: the test must make the faulty policy produce an incorrect response which is different from the expected response that should be produced by the correct policy.

We then exploit FDC to formulate strong mutation-based test generation.

1.3.2 Strong Mutation-based Test Generation with Fault Detection Conditions

Strong mutation-based test generation involves generating test input that satisfies the three constraints of the fault detection conditions - reachability, necessity and sufficiency/propagation for each fault type. If we use only reachability and necessity constraint to generate test suites, then it is called weak mutation-based test generation and if we use all three constraints, then it is referred to as a strong mutation. Strong mutation assures fault detection while weak mutation could not. However, incorporating propagation constraint is costly and requires a lot of effort that becomes infeasible to apply strong

mutation [20]. As a result, weak mutation is used in many cases, however, the features of XACML (being the domain specific language) made it feasible to apply strong mutation. In fault detection conditions, we specify all possible mutually exclusive constraints for reachability, necessity and propagation but for mutation-based test generation, we need not require generating test cases to satisfy all possible cases. For example, if reachability constraint is "policy target evaluates to *true*" or "policy target evaluates to an error", we may just use "policy target evaluates to *true*" for fault detection and avoid "policy target evaluates to an error" to avoid redundant tests and for simplicity. Hence, we identify just sufficient mutually exclusive conditions from each of the reachability, necessity and propagation constraints for the fault to be detected. The process of identifying sufficient constraints to detect a fault involves picking one of the mutually exclusive constraints from the reachability constraint and concatenating the reachability constraint with corresponding mutually exclusive necessity constraints and propagation constraints. For example, if the reachability constraint has two mutually exclusive conditions which are "policy target evaluate to *true*" or "policy target evaluate to an error", we can pick "policy target evaluates to *true*" as the reachability constraint. Further, the chosen reachability constraint may have two mutually exclusive necessity constraints as "rule target evaluates to *true*" or "rule target evaluates to an error". Hence, we concatenate one of the mutually exclusive necessity constraints, say "rule evaluates to *true*". Finally, we concatenate one of the mutually exclusive propagation constraints corresponding to the chosen necessity constraint, say it is "all rules with deny effect except the first rule should not evaluate to *true*". Hence, the sufficient constraint to identify the fault is "policy target evaluates to *true* ∧ rule target should evaluate to *true* ∧ all deny rules except the first rule should not evaluate to *true*".

The test suite generated in this way is called a mutation-based test suite. The resulting test suite may have many redundant test cases which will only kill those mutants which will be killed by other test cases. In other words, redundant test cases do not kill unique mutants that are not killed by any other test cases. Hence, such redundant test cases do not contribute to fault detection capability and result in poor cost-effectiveness of a test suite. Since our goal is to generate a near-optimal test suite, we need to optimize the test suite. Let, M_i represents the set of mutants killed by an arbitrary test case t_i. By near-optimal, we mean, if $T = \{t_1, t_2, ..., t_n\}$ be a near-optimal test suite, then for any arbitrary i and j, such that $i != j$, $M_i - M_j != empty$ as well as $M_j - M_i != empty$ i.e each test in a test suite kills at least one mutant not killed by any other test cases. The reason for generating a near-optimal test suite is that it can be used for evaluating the cost-effectiveness of a test suite from other testing methods. Hence, we have applied the optimization to find the near-optimal test suite.

Once we have the near-optimal test cases, we need Oracle value (expected response). Since we have the original policy which we assume to be correct, we run generated test inputs on original policy and record its value as Oracle values for a test suite. The optimized version of strong mutation-based test generation is named as SMT. The optimization involved in SMT is a costly operation making it infeasible to apply for large policies. We also formulate strong mutation-based test suite without optimization referred to as NO-SMT which has less cost-effectiveness as that of SMT but achieves perfect mutation score.

1.3.3 Quantitative Analysis

Finally, we do quantitative analysis using metrics like mutation score and mutants killed per test (MKPT) to determine the cost-effectiveness of major testing methods of XACML (such as rule coverage, decision coverage, non-error decision coverage, MC/DC coverage and non-error MC/DC coverage). Mutation score is the percentage of mutants killed against the number of non-equivalent mutants. It indicates the fault detection capability of a test method. MKPT is the average number of mutants killed by a test in a test suite. An optimal test suite will have the highest MKPT score. As a result, we can evaluate the cost-effectiveness of a testing method by comparing the MKPT score of the test suite from the method under consideration with an optimal test suite. However, finding an optimal test suite is an undecidable problem and hence, we used the SMT with the approximation method to optimize the test suite to find near-optimal test suite. The goal is to compare the MKPT score of a near-optimal test suite with the test suite from the current method and establish the cost-effectiveness of the testing method.

The rule coverage is the coverage criteria which aims to evaluate the effect of each rule in the policy. The decision coverage is the coverage criteria which aims to evaluate each decision point (policy set target, policy target, rule target and rule condition) to three possible evaluations *true*, *Non-applicable (N/A)* and *error*. Non-error decision coverage is the same as decision coverage except that it does not consider the *error* in evaluation of decision points i.e it aims to evaluate each decision point to only *true* and *N/A* [27]. Consider, an expression "resource-id=Liquor ∨ resource-id= Medicine" which is composed of a disjunction of two constraints "resource-id=Liquor" and "resource-id= Medicine". A coverage criterion which satisfies decision coverage and in addition requires

that every condition in a decision point has taken on all possible outcomes at least once and each condition has been shown to independently affect the decision's outcome is referred to as MC/DC coverage criteria. For example, MC/DC of a conjunctive expression with n conditions *(e.g.,* $c_1 \wedge \ldots \wedge c_n$*)* requires $n+1$ tests: one test that evaluates all conditions to *true* and *n* tests that evaluate one condition to *false* and other conditions evaluate to *true*. MC/DC of a disjunctive expression with *n* conditions *(e.g.,* $c_1 \vee \ldots \vee c_n$*)* requires $n+1$ tests: one test that evaluates all conditions to *false* and n tests that evaluate one condition to *true* and other conditions evaluate to *false* [27].

1.4 Outline

The remainder of this document is organized as follows. Chapter 2 presents the background and summarizes related work on the research topic. Chapter 3 specifies the Fault Detection Condition (FDC) for the fault model. Chapter 4 presents the mutation-based test generation, Chapter 5 specifies the quantitative evaluation, and Chapter 6 concludes this work.

CHAPTER 2

Background and Related Work

2.1 Mutation analysis and mutation-based test generation

Mutation analysis is a fault-based testing technique which provides a testing criterion that can be used to measure the effectiveness of a test set. The general principle of Mutation analysis is to produce mutants of the original source/specification by injecting the mistakes that programmers/users might make. Such faulty programs/specification resulted after deliberately seeding faults into the original source are called mutants. The statement in which mutation takes place is called a mutation point and the transformation rules used to produce such fault and hence mutants are called mutation operators. The resulting faulty policies (mutants) which exhibit the same behavior as the original ones are known as equivalent mutants and those which exhibit different behavior than that of the original ones are called nonequivalent mutants [19].

Mutation operators are defined with respect to a fault model, which is a collection of the fault types in the programming language. The main hypotheses of mutation testing [19, 32] include: (a) the mutants are based on actual fault models and are representative of real faults, (b) developers produce programs (policies) that are close to being correct, (c) tests sufficient to detect simple faults (i.e., in mutants) are also capable of detecting complex faults. Experiments have shown that mutants are indeed similar to real faults for the purpose of evaluating testing techniques [16, 33]. To assess the quality of a test set, the

generated mutants are run against the test set. If the result from a mutant is different from the original one for any test cases in the input test set, we say that the mutant failed the test. A mutant is said to be killed if it fails one or more tests. One outcome of the mutation testing process is the mutation score – mutant killing ratio. Mutant-killing ratio is the ratio between the number of mutants killed by the test suite against the total number of non-equivalent mutants [3].

Mutation-based test generation derives a test from one or more mutants of a given program so that the mutant and its original program produce a different execution result. Such a test needs to meet the following constraints: (a) Reachability constraint: the test must reach the mutation point, i.e., trigger the execution of the mutated code, (b) Necessity constraint: the test must make the mutated code evaluate to an intermediate result that is different from that of the original program, and (c) Propagation (a.k.a. sufficiency) constraint: the test must make the intermediate result of the mutated code propagate to a final state that is different from the final state of the original program. We refer to the collection of reachability, necessity, and propagation constraints as the fault detection condition. The existing techniques primarily follow the concept of weak mutation testing [20] that uses the reachability and necessity constraints to generate test inputs [31]. The main reason is that it has been shown to be intractable to solve the propagation constraint [28]. This paper aims at strong mutation testing of XACML policies that deal with all reachability, necessity, and propagation constraints. As a domain-specific language, XACML has a unique structure that makes it feasible to tackle propagation constraints. The existing work on mutation testing of XACML policies focuses on mutation tools and

evaluation of testing methods with policy mutants. In comparison, this paper focuses on the formalization of the fault detection conditions and mutation-based test generation.

2.2 Introduction to XACML

XACML is a general-purpose access control policy language. The root of an XACML policy document $\mathbb{P}$ is a policy element or a policy set element. A policy set element contains other child policy elements or policy set elements such that each of which may evaluate to different access control decisions. A policy element contains a list of the rule elements each of which may evaluate to different access control decisions. And, there is a mechanism to resolve decisions from multiple units of the rules within a policy or multiple units of policies or policy sets within a policy set known as combining algorithms. The combining algorithm which reconciles decisions from a list of the rules is referred to as rule combining algorithm. Similarly, the combining algorithm which reconciles decisions from policies or policy sets is referred to as policy combining algorithm. A policy set, policy or rule element contains a target element that specifies the set of requests to which it applies. Further, a rule may consist of another Boolean function known as condition element which needs to be satisfied for a rule to be applied. Figure 2.1 depicts the XACML policy language model.

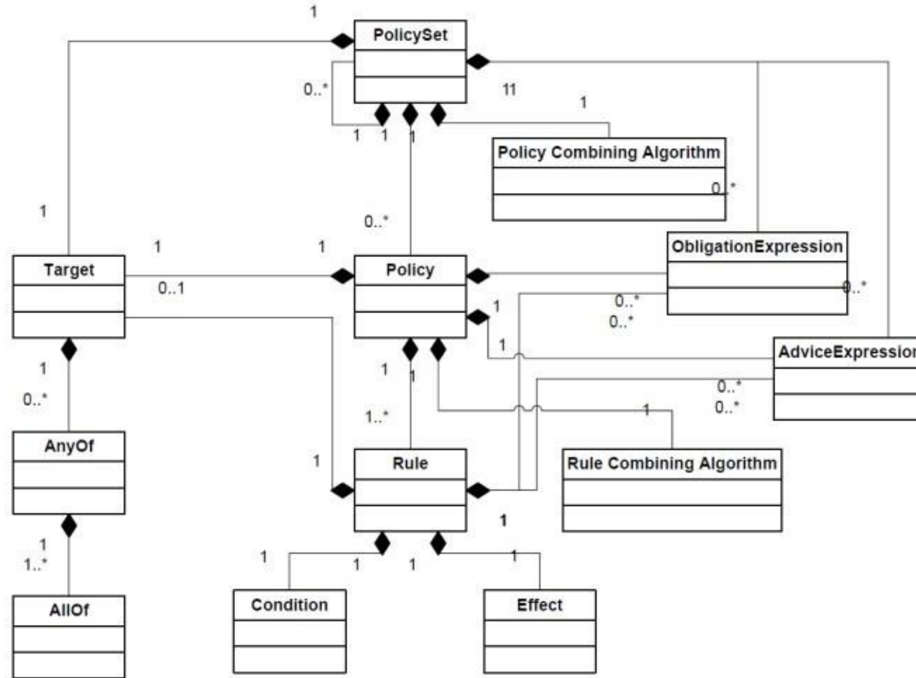

Figure 2.1: XACML policy language model [2]

As shown in Figure 2.1, the target of a rule, policy, or policy set is a conjunctive sequence of *AnyOf* clauses. Each *AnyOf* clause is a disjunctive sequence of *AllOf* clauses, and each *AllOf* clause is a conjunctive sequence of match predicates. A match element matches and compares attributes in a request context with the embedded attribute values. Logical expressions for match predicates and rule conditions are usually defined on four categories of attributes: subject, resource, action, and environment. They can use a great variety of predefined functions and data types. A rule also has an effect element which will be either permit or deny corresponding to the access decision of the rule.

Formally, a policy set element PS is a quintuple $< PST, PCA, [P_1, P_2, \ldots, P_m], A, O>$, where PST is the policy set target, PCA is the policy combining algorithm, and $[P_1, P_2, \ldots, P_m]$ is the list of policies or policy sets in the policy set, A is a set of advice, and O is a set of Obligation. Each policy P_i is a quintuple $<PT_i, RCA_i, [r_{i1}, r_{i2}, \ldots, r_{in}], A_i, O_i>$, where PT_i is the policy target, RCA_i is the rule combining algorithm, and $[r_{i1}, r_{i2}, \ldots, r_{in}]$ is the list

of the rules in the policy, A_i is a set of advice, and O_i is a set of obligation. Each rule R_j is a triple $< rt_j, rc_j, re_j >$, where rt_j is the rule target, rc_j is the rule condition, and $re_j \in \{permit, deny\}$ is the rule effect. $< rt_j, rc_j, permit>$ is called a permit rule, whereas $< rt_j, rc_j, deny>$ is a deny rule. If both rt_j and rc_j are omitted (always *true*), then the rule $< _, _, re_j >$ is a default rule. More specifically, $< _, _, deny>$ is a default deny rule, whereas $< _, _, permit>$ is a default permit rule.

To access the resource, a subject presents an access request to the system. An access request for an ABAC authorization system consists of a set of attributes. For an access request q, a policy or policy set responds with an access decision, such as permit or deny. Given an access request q, PS is evaluated to produce a response (i.e., access decision) denoted as $d(PS, q)$. A policy set target PST is first evaluated according to the attribute values in q. If the result of the evaluation is *false*, then $d(PS, q) = N/A$ otherwise policies $P_1, P_2, \ldots$, and P_m will be evaluated if PST is *true* or evaluates to an error. $d(PS, q)$ depends on policy combining algorithm PCA and the decisions of individual policies with respect to q (denoted as $d(P_i, q)$). Similarly, for an individual policy $P_i = <PT_i, RCA_i, [r_1, r_2, \ldots, r_n] >$, policy target PT_i is evaluated according to the attribute values in q. If the evaluation result is *false*, then $d(P_i, q) = N/A$, otherwise, rules $r_1, r_2, \ldots$, and r_n will be evaluated. $d(P_i, q)$ depends on rule combining algorithm RCA_i and the decisions of individual rules. Decision of the rule $r_j = < rt_j, rc_j, re_j >$ with respect to q, denoted as $d(r_j, q)$, is defined as follows:

i) *Permit*: access is granted when re_j = permit and rt_j and rc_j is *true* with respect to q.

ii) *Deny*: access is denied when re_j = deny, and rt_j and rc_j is *true* with respect to q.

iii) *Non-applicable*, or simply *N/A: q* is not applicable, i.e., rt_j and/or rc_j is *false* with respect to *q*.

iv) *IndeterminateD* or simply *I(D)*: An *error* occurred when rt_j or rc_j was evaluated and $re_j = Deny$. The decision could have evaluated to Deny if no *error* had occurred. A syntactically valid access request may cause the occurrence of a runtime *error* for different reasons, such as missing an attribute value, mismatch of an attribute type, and an exception of expression and function evaluation.

v) *IndeterminateP* or simply *I(P)*: An *error* occurred when rt_j or rc_j was evaluated and $re_j = Permit$. The decision could have evaluated to Permit if no *error* had occurred.

A rule may have an empty target as well as empty conditions referred to as a default rule. For a default rule $r_j = < _, _, re_j >$, any access request *q* is $d(r_j, q) = re_j$.

The root element of a general XACML policy document $\mathbb{P}$ could be either a policy element or a policy sets element. If the root element is a policy set element, then policy combining algorithms for the root policy set has nested rule combining algorithms and/or policy combining algorithms inside it. Since we need to deal with five rule combining algorithms in our work and if we consider policy sets we need to consider six policy combining algorithms, such nesting would create lots of combinations of nested combining algorithms. As a result, for simplicity, further, in this work, we would only consider XACML policy document $\mathbb{P}$ which has a policy element as a root of the XACML document so that we don't need to deal with nested rule and policy combining algorithms. With similar reasoning, we can also deal with policy sets and hence policy combining algorithms but for simplicity, we only consider policy in this work. Further, Advice and Obligation plays no role in our work, so we omit them while representing policy elements further in

this work. Hence, for simplicity, we represent XACML policy document $\mathbb{P}$ as P = <*PT, RCA, [r_1, r_2, ..., r_n]*> where *PT* is policy target, *RCA* is rule combining algorithm and *[r_1, r_2, ..., r_n]* is the list of child rules of *P*.

2.2.1 Sample Policy

Figure 2.2 presents an example of an XACML policy document which has Policy Id *KmarketBluePolicy*. It is a demonstration policy from Balana – Open source implementation of XACML [13]. The rule combining algorithm of the policy is *deny-overrides* (line 2). The policy's target (lines 3-14) implies the constraint "*role=blue*" where a role is an attribute in the subject category and blue is the value for the attribute of type string. For this policy to be applied to a request, the request context must contain the subject attribute role with the value of blue.

There are three rules with rule ids: deny-liquor-medicine (line 16-37), max-drink-amount (lines 38-61), and permit-rule (line 62). The target of the rule deny-liquor-medicine (lines 18-36) implies the constraint "*resource-id=Liquor*" (line 19-26) ∨ "resource-id=Medicine" (lines 27-34), where resource-id is an attribute in the resource category. Since the rule does not have a condition element, it is *true* by default, hence, the rule will result in a "Deny" decision if "resource-id=Liquor ∨ resource-id=Medicine". The target of the rule max-drink-amount implies the constraint "resource-id=Drink", and the condition the implies constraint "amount > 10". Thus, the rule results in a deny decision if "resource-id=Drink ∧ amount > 10". Rule permit-rule has neither target nor condition. It results in a Permit decision whenever it is reached.

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

Figure 2.2. A sample XACML policy [13]

Hence, in the notation we described earlier, P = <*role* =*"blue",deny-overrides* , [r_1,r_2,r_3]> *where* r_1 = <rt_1, rc_1, re_1>, r_2 = <rt_2, rc_2, re_2>, r_3 = <rt_3, rc_3, re_3> such that rt_1 = "resource-id = Liquor ∨ resource-id = Medicine", rc_1 = *true* (since its empty), re_1 = *deny,* rt_2 = "resource-id = Drink", rc_2 = *"amount > 10"*, re_2 = *Deny,* rt_3 = *true* (since its empty), rc_3 = *true* (since its empty), and re_3 = *Permit.*

To illustrate how an XACML authorization scheme works, we first need to discuss how each element of XACML evaluation occurs.

2.2.2 Policy Evaluation

When a request is presented to the AC system, it first needs to determine whether the available set of XACML policies can be applied to a given request or not. For this purpose, the target element *PT* of root policy element *P* of XACML policy document $\mathbb{P}$ is used. If it is empty, the policy is applicable to any request *q*. If it is not empty and request *q* meets the constraints specified by the target *PT* or if there is an *error* while evaluating the target *PT*, the policy is applicable to request *q* otherwise policy is not applicable and will not be evaluated further. The evaluation of policy involves evaluation of its child rules. Once its child elements are evaluated and it obtains the authorization decision from each child element, it uses a rule combining algorithm to reconcile the decision obtained from various child rules and makes a final authorization decision from the result of the rule combining algorithm [2]. The details on combining algorithms is presented in Section 2.2.5. If policy target *PT* evaluates to *true*, the policy-level decision will be the decision from the rule combining algorithm itself. However, if *PT* evaluates to an error, the decision will be made as described in Section 2.2.3

2.2.3 Policy Evaluation and Indeterminate Target

If the target of a policy evaluates to an error, the result of policy evaluation will be *N/A* whenever the result of the rule combining algorithm is *N/A*; the result will be *I(P)* whenever the result of the rule combining algorithm is permit; the result will be *I(D)* if the result of the rule combining algorithm is deny, and the result will be *I(DP)* if result of combining algorithm is Indeterminate. For any other indeterminate result *{I(DP), I(D), I(P)}* from the rule combining algorithm, the result for policy evaluation is the same as that for the rule combining algorithm. It uses a rule combining algorithm to reconcile decisions obtained from various child rules and makes a final authorization decision from the result of the rule combining algorithm [2].

2.2.4 Rule Evaluation

The evaluation of ith rule r_i occurs if the evaluation of the rules above it does not halt the evaluation of policy *P* producing the result *d(P,q)*. The rule r_i is applicable to request *q* if "rule target rt_i, as well as rule condition rc_i, is *true*" or "rule target rt_i evaluates to an error" or "rule target rt_i is *true* and rule condition *evaluates to an error*" [2]. We say rule evaluates to *true*, if rule target, as well as rule condition, evaluates to *true*. Similarly, if rule target or rule condition evaluates to *false*, we say rule evaluates to *false* or *N/A*. If rule target or rule condition evaluates to an error, we say rule evaluates to an *error*.

If a rule evaluates to *true*, the rule-level decision ($d(P,r_i)$) will be the effect of the rule i.e if the effect of the rule is *permit*, then the decision of the rule will be *permit*. If a rule evaluates to an error, the rule-level decision will be the Indeterminate of corresponding effect i.e if the effect is *permit*, the decision of the rule will be *I(P)* or simply *I(P)*.

For a request *q*, there may be multiple rules which are applicable such that each produces their effect and it is the job of the rule combining algorithm (RCA) to reconcile decision of multiple rules and produce the final decision for the policy evaluation. Section 2.2.5 discusses various rule combining algorithms we considered in this work.

Let *rca(P, q)* denote the result of applying *RCA* to the rules in *P* for the request *q*. Assuming that the list of the rules in *P* is non-empty, *rca(P, q) ∈{Permit, Deny, N/A, I(D), I(P), I(DP)}* and per the standard specification [2], *d(P, q)* is defined as follows:

$$d(P,q) = \begin{cases} N/A, & \neg PT \\ rca(P,q), & PT \\ rca(P,q), & Error(PT) \wedge rca(P,q) \in \{N/A, I(D), I(P), (DP)\} \\ I(P), & Error\ (PT) \wedge rca(P,q)\ =\ Permit \\ I(D), & Error\ (PT) \wedge rca(P,q)\ =\ Deny \end{cases}$$

2.2.5 Combining algorithms

There are eleven RCAs in XACML 3.0. Four of them are for compatibility support for older versions - *legacy ordered-deny-overrides*, *legacy deny-overrides*, *legacy ordered-permit-overrides*, and Legacy *ordered-permit-overrides* [2]. As they are for the backward compatibility for the previous version of XACML, we do not consider them in our work. Among the remaining seven, the five are listed below and the other two are *ordered deny-overrides* and *ordered permit-overrides*. In Balana [13] (an open source implementation of XACML3.0), the implementations of *ordered-deny-overrides* and *ordered-permit-overrides* are the same as *deny-overrides* and Permit-overrides. As a result, we do not consider these two as well and this work only focuses on the five RCAs which are as follows:-

a) *deny-overrides: deny-overrides* is intended for those cases where a deny decision should have priority over a Permit decision. If any rule evaluates to *deny*, the result is *deny*. If there is no *deny* decision from any rules and if any decision is *I(DP)*, the result is *I(DP)*.

If there is no *deny* and *I(DP)* decision, and if any decision is *I(D)* and another decision is *I(P)* or deny, the result is *I(DP)*. If it is also not *true*, then if any decision is *I(D)*, the result is *I(D)*. If there is no *deny*, *I(DP)* and *I(D)* decision, and if any decision is *permit*, the result is *permit*. If it is also not *true* and if any decision is *I(P)*, the result is "*I(P)*" otherwise, the result is *N/A* [2].

b) *permit-overrides: permit-overrides* RCA is intended for those cases where a permit decision should have priority over a deny decision. If any decision is *permit*, the result is *permit*. If there is no *permit* decision and if any decision is *I(DP)*, the result is *I(DP)*. If there is no *permit* and *I(DP)* decision, and if any decision is *I(P)* and another decision is *I(D)* or *deny*, the result is *I(DP)*. If it is also not *true*, then if any decision is *I(P)*, the result is *I(P)*. If there is no *permit*, *I(DP)* and *I(P)* decision, and if any decision is *deny*, the result is *deny*. If it is also not *true* and if any decision is *I(D)*, the result is *I(D)* otherwise, the result is *N/A*.

c) *deny-unless-permit:* This is intended for those cases where a *permit* decision should have priority over a *deny* decision, and an *Indeterminate* or *N/A* must never be the result if the policy is applicable to the request. If any decision is *permit*, the result is *permit* otherwise, the result is *deny*.

d) *permit-unless-deny:* This RCA is intended for those cases where a *deny* decision should have priority over a *permit* decision, and an *Indeterminate* or *N/A* must never be the result. If any decision is *deny*, the result is *deny* else the result is *permit*.

e) *first-applicable:* This RCA is intended for those cases where the evaluation of policy should halt as soon as any rule is applicable to request *q*. Rules are evaluated in the order in which they are listed. If a rule's target matches and condition evaluates to *true*,

then the result is rule's effect (*permit* or *deny*). If a rule's target evaluates to an error or rule condition evaluates to an error, then the result is *I(P)* if rule's effect is *permit* or *I(D)* if rule's effect is *deny*. If the target or condition evaluates to *false*, the next rule is evaluated. If no further rule exists, then the result is *N/A*.

To illustrate how XACML policy evaluation occurs, the following section presents sample requests and discusses how the policy evaluation takes place for the given sample policy.

2.3 Sample Requests and Policy Evaluation

This section presents some sample requests and how the evaluation occurs in sample policy. The XACML request contains a list of attributes and their value. Let us consider q1 be the first sample request as q_1 = {*resource-id =”Liquor”*}. This request consists of one attribute-value pair where *resource-id* is attribute and *Liquor* is a value of the attribute. Since the policy target of sample policy is "*role = blue*" and q_1 does not contain role attribute, the given sample policy will not be applicable to q_1 because the policy target did not match, and no rules will be evaluated, and the final decision will be *N/A*.

Let us consider second request q_2 as {*resource-id =”Liquor” and role = “gold”}*. Since q_2 does contain role attribute but it is not *blue*, the policy will not be applicable because the policy target did not match, and the final decision of the policy evaluation will be *N/A*.

Let us consider a third request q_3 as {*resource-id =”Liquor” and role = “blue”}*. Since q_3 does contain role attribute and is *blue* the policy will be applicable. Further, it also consists of another attribute-value pair *resource-id = ”Liquor”*. As a result, rule 1 is applicable whose effect is *deny* and since the RCA is *deny-overrides*, the rule evaluation

stops after rule1 and decision of RCA will be *deny*. Since the PT evaluates to *true*, the final decision for the policy evaluation will be *deny*. However, if PT evaluated to an error, the final decision for policy would be *I(D)*.

Let us consider a fourth request *q4* as {*resource-id = "aFFFF" and role = "blue"*}. Since role attribute is *blue,* the policy will be applicable. The value of another attribute *resource-id* is *aFFFF*. As a result, rule 1 is not applicable as well as rule 2 because their target didn't match. However, rule 3 is a default rule with no target and condition, so it will be applicable to any request and produces *permit* effect. Since the rule level decision of the first two rules are *N/A* and that of the third rule is *permit*, the RCA level decision will be *permit*. Since *PT* evaluates to *true*, the final policy-level decision will be *permit*.

2.4 Related Work

A test for an XACML policy consists of a test input and the corresponding Oracle value (i.e., expected response to the access request). Oracle values depend on the access control requirements of the system under test. A test fails when the system's actual response to the request is different from the expected response. Such a failure often indicates the existence of a fault that may lead to unauthorized access, elevated privilege, or denial of service. The existing approaches to test generation for 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. This paper is mostly related to the work that generates test inputs from the XACML policy under test.

The existing testing methods for XACML policies generate access requests directly from the policy under test. A user needs to define the expected response for each request to determine whether each test passes or fails. Martin et. al. generates access requests in Cirg from counterexamples produced by model checker Margrave [21] through the change-impact analysis [10]. Mutation score of the testing methods in Cirg ranged from 30% to 60% in different case studies and 100% for a simple policy. Targen [22] obtained mutation score that ranged from 75% to 79% for different case studies which derive access requests to satisfy all the possible combinations of truth-values of the attribute id-value pairs found in a given policy [5]. The X-CREATE framework deals with the structures of the Context Schema Considering that requests must conform to the XML Context Schema. Bertolino et al. have developed the Mutant-killing ratios of the X-CREATE framework ranging from 75% to 96% for several small policies [7]. They have also developed other test selection strategies, such as Simple Combinatorial and Incremental XPT [6].

Mutant-killing ratios of the Simple Combinatorial strategy ranged from 3% to 100%, whereas mutant killing ratios of the Incremental XPT strategy ranged from 55% to 100%. Bertolino et al. [8] 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. Mutant-killing ratios of this approach ranged from 62% to 98%. In addition, Bertolino et al. [9] proposed similarity-based metrics for prioritizing existing tests of policies. This work is not concerned with how the tests are generated, though. Li et al. [12] have developed XPTester, which used symbolic execution technique to generate requests from XACML policies. They convert the policy under test into semantically equivalent C Code Representation (CCR) and

symbolically execute CCR to create test inputs and translate the test inputs to access requests. Mutant-killing ratios of XPTester ranged from 37% to 93%. Although all the above work uses mutation to evaluate fault detection ability, there are subtle differences between the subject policies and the fault models used by different research groups. It is obvious that the above methods are far from satisfactory for the high assurance of XACML policies. Most of them produce many tests by combining attribute values. None of them have considered advanced coverage criteria, e.g., decision coverage and MC/DC, for access control constraints (i.e., rule target, rule condition, policy target, and policy set target).

Verification techniques have also been proposed for quality assurance of XACML policies. The verification system in Margrave checks whether an XACML policy satisfies given properties that describe the constraints on attributes. Margrave transforms the XACML policy into multi-terminal binary decision diagrams. Hwang et. al. [23] applied Margrave to the detection of multiple-duty-related security leakage. Hughes and Bultan [25] developed an approach for defining properties as partial orderings between XACML policies, translating them to Boolean formulas, and using the Zchaff SAT solver to check satisfiability of the Boolean formulas. Hughes and Bultan have also proposed an approach for translating XACML policies into the Alloy language and analyzing properties as partial ordering relations [24]. The above verification techniques are premature because they only deal with a very restricted subset of XACML (e.g., no or limited attribute data types and no complex conditionals). In addition, they require formal representation of application-specific properties, which can be a non-trivial task for XACML users.

CHAPTER 3

Fault Detection Condition

A fault in a policy is an *error* or flaw that can make it produce an incorrect result. Consider, the ith rule of a policy P should have permit effect, but if it is somehow changed to deny, then there is a fault in the effect of a rule. Since the effect of a rule in a policy is incorrect, we refer to such a fault as an Incorrect Rule Effect fault. Similarly, if there is a fault in the target element of a rule, then it is referred to as an Incorrect Rule Target fault. Incorrect Rule Condition fault represents the fault in the condition element. If a rule is missing, we refer to such a fault as a missing rule fault. If there is a different combining algorithm than it is supposed to have, then it is an called Incorrect Rule (Policy) Combining Algorithm fault.

Table 3.1: Fault Model

Mutation Operator			**Fault Type**
No	**Name**	**Meaning**	
1	CRE	Change Rule Effect	Incorrect Rule Effect
2	RTT	set Rule Target True	Incorrect Rule Target
3	RTF	set Rule Target False	
4	RCT	set Rule Condition True	Incorrect Rule Condition
5	RCF	set Rule Condition False	

6	ANF	Add Not Function in condition	
7	RNF	Remove Not Function in condition	
8	RER	REmove a Rule	Missing Rule
9	FPR	First Permit Rule	Incorrect Rule Ordering
10	FDR	First Deny Rule	
11	PTT	set Policy Target True	Incorrect Policy Target
12	PTF	set Policy Target False	
13	RPTE	Remove Parallel Target Element	Missing target element
14	CRC	Change Rule Combining Algorithm	Incorrect Combining Algorithm

Similarly, if there is a fault in the ordering of the rules, then it is referred to as an Incorrect Rule Ordering fault. If either Match element, *AnyOf* element or *AllOf* element of Target element is missing, we refer to such a fault as a Missing Parallel Target Element fault. If there is a fault in policy target, then we refer to such a fault as an Incorrect Policy Target fault.

While defining policy, there could exist various faults. The policy with the fault is referred to as a faulty policy. If P' is a fault policy of original policy P, then P' is nothing but the result of an application of some form of transformation rule that introduces the fault. For example, the transformation rule could be "change rule effect" which alters the

effect of the rule from permit to deny and vice versa. In mutation analysis literature, the transformation rule which results in the faulty policy from the correct one is referred to as a mutation operator and the faulty policy itself is referred to as a mutant. Table 3.1 consists of fourteen mutation operators which are categorized into eight faults types. Mutation operators are defined with respect to a fault model, which is a collection of the fault types in the given domain of programing language or specification. The details on each mutation operator in Table 3.1 are discussed in Sections 3.1 through 3.14.

The fault in a policy may result in it to produce an incorrect result i.e. it results in a different output than it is supposed to produce. We can exploit this difference in result to reveal a fault by supplying the same input to both original policy and mutant, and if there is a difference in response, then we could conclude that there exists a fault. A fault policy, however, does not necessarily produce a different result than the correct policy for all inputs. The test inputs must satisfy certain conditions to produce a different result than from the correct policy and hence reveal the fault. Such conditions are referred to as fault detection conditions.

Hence, the fault detection condition (FDC) of a given fault should specify the constraints that a test case must satisfy to reveal the fault. As discussed in Section 1.3.1, the fault detection condition can be formulated with three constraints which are reachability constraint, necessity constraint and propagation constraint.

a) *Reachability(**R**) constraint*: Reachability constraint specifies that the test must reach the faulty policy element (e.g., rule target, rule condition, rule effect, policy target, and combining algorithm). If the faulty element is not evaluated, then the faulty policy will behave the same as that of the original policy and we cannot distinguish the fault. The first

rule of a sample policy in Figure 2.2 of Chapter 2 has deny effect with id "deny-liquor-medicine". Consider there is a faulty policy which is same as the original policy except for the effect of the first rule with id "deny-liquor-medicine" has permit effect instead. To identify this fault, this rule must be evaluated. To evaluate the rule, the rule should be reached during policy evaluation. To reach the first rule, the policy target must evaluate to *true* or *error*. Hence, the reachability constraint here is to evaluate policy target to *true* or *error*.

b) *Necessity(**N**) constraint*: The test must make the faulty element evaluate to an incorrect intermediate result, which is different from the evaluation result that should be produced by its correct counterpart. For example, for the fault considered in reachability constraint above, the rule target and/or condition of the faulty rule should evaluate to an error or *true* so that it produces an incorrect intermediate result. Hence, the necessity constraint is "rule target and rule condition of the faulty rule does not evaluate to *N/A*" (i.e. either rule target and condition evaluates to *true* or rule target evaluates to an error or rule target evaluates to *true* and rule condition evaluates to an error).

c) *Propagation(**P**) constraint*: The test must make the faulty policy produce an incorrect response, which is different from the expected response that should be produced by the correct policy. For example, to propagate incorrect intermediate results for revealing the fault specified in reachability constraint, all the other rules with deny effect except the first rule should not evaluate to *true* i.e. either they should evaluate to an error or *false* because the rule combining algorithm is *deny-overrides* and if any another deny rule evaluates to *true*, it will produce the deny result in both faulty and correct policy.

We use the notation *PT*, $\neg PT$ and *Error (PT)* to denote that policy target evaluates to *true*, *N/A* and *error* respectively. We mention rule evaluates to *true* if both target and condition of the rule evaluates to *true*. We use the expression $rb_i = (rt_i \wedge rc_i)$ to denote that the target, as well as condition of ith rule, evaluates to *true*. Similarly, $Error(rb_i)$ denotes an *error* in the evaluation of the target or condition of the rule r_i. Further, $\neg rb_i$ denotes either rt_i or rc_i evaluates to *false* such that rule is *N/A*. Additionally, if none of the attributes in the request context matches the attributes in the rule target or rule condition, then the rule is not applicable for the request. We use the notation *I(P)*, *I(D)* and *I(DP)* to denote *Indt*, *I(P)* and *Indeterminate* respectively. We use the notation $r_i = <rb_i, re_i>$ whenever possible to represent i^{th} rule $<rt_i, rc_i, re_i>$ for simplicity. Further, we interchangeably use notation r_i or *current rule under consideration* to denote an arbitrary i^{th} rule.

3.1 FDC for Change Rule Effect (CRE)

Change Rule Effect (CRE) is a mutation operator for the incorrect rule effect fault type in which there is a fault in effect of a rule element. Since there are only two possible rule effects - permit or deny, there will be a fault in effect of a rule if the effect of a rule gets altered from permit to deny and vice-versa. The flipping of the rule effect is the only mutation operator for incorrect rule effect fault.

Consider an XACML policy P = <PT, RCA, RL> where PT is Policy Target, RCA is Rule Combining Algorithm and RL = [r_1, r_2, …, r_n] is a list of the rules. If the effect of ith rule r_i is flipped to deny by some incident, then the resulting policy will be P’ as shown in Table 3.2. Since ith rule is supposed to have permit effect but in P’ it is *deny*, so P’ is the faulty policy. Here, P’ is called the Change Rule Effect (CRE) mutant of P.

Table 3.2: A Faulty Policy with an Incorrect Rule Effect

	Correct Policy P		***Faulty Policy P'***	
Policy target	*PT*		*PT*	
Rule combining algorithm	*Permit-Overrides*		*Permit-Overrides*	
Rules *R*	r_1	$<rb_1, re_1>$	r_1	$<rb_1, re_1>$
	…	…	…	…
	r_i	$<rb_i$, ***Permit***$>$	$r_{i'}$	$<rb_i$, ***Deny***$>$
	…	…	…	…
	r_n	$<rb_n, re_n>$	r_n	$<rb_n, re_n>$

Since the fault detection condition depends on the rule combining algorithm (RCA) of a policy, we present fault detection condition for each of the RCA.

a) Permit-overrides:

i) <u>Reachability constraint</u>

The reachability constraint must trigger the evaluation of the rule with faulty effect i.e. it should result in the evaluation of ith rule in both *P* and *P'*. The rules in a policy will only be evaluated if the policy target is *true or* evaluates to an *error*. Further, when the rule-combining algorithm is *Permit-overrides*, rule r_i will not be triggered if there is a *permit* rule before rule r_i that evaluates to a *permit* decision. Thus, the reachability constraint is - for any *permit* rule r_j $(j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e. it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = \langle rb_j, permit\rangle$ *for* $j < i\}$

ii) <u>*Necessity constraint*</u>

The necessity constraint should make i^{th} rule r_i in P and rule r_i' in P' produce a different rule-level decision. This requires that rb_i should evaluate to either *true* so that the ith rule decision will be *permit* in P and *deny* in P' respectively or should *evaluate to an error* so that rule decision will be $I(P)$ in P and $I(D)$ *in* P'. If rb_i evaluates to *N/A*, the rule level decisions will be *N/A* in both P and P' in which case we cannot distinguish the faulty policy. Hence, formally, necessity constraint is $rb_i \vee Error(rb_i)$.

iii) <u>*Propagation constraint*</u>

Given a faulty element produces different intermediate results, the *propagation constraint* must make P and P' produce a different policy-level decision. In other words, the different intermediate result from necessity constraint should contribute to producing a different policy-level decision. For this, any *permit* rule r_j $(j>i)$ after r_i, r_j should not evaluate to a *permit* decision, otherwise, $d(P, q)= d(P', q) = Permit$. The test must make rb_j evaluate to *N/A* or *error* for each *permit* rule r_j $(j>i)$. Therefore, the *propagation constraint* can be formalized as $\neg rb_j \vee Error(rb_j)$ for any rule $r_j = \langle rb_j, Permit\rangle$ $(j>i)$. Let, Þ = "$\neg rb_j \vee Error(rb_j)$ for any rule $r_j = \langle rb_j, Permit\rangle$ $(j>i)$". Þ is sufficient for propagation if necessity constraint is - rb_i evaluates to *true*. However, when rb_i evaluates to an error, Þ is not sufficient for propagation.

We can verify that *when* rb_i *evaluates to true*, Þ is sufficient for propagation by showing that it holds for all possible evaluation of other rules (all the rules in the policy except the current rule under consideration) as shown in Table 3.3. Let, the ith rule in the

correct policy *P* has *permit* effect, then that of the faulty policy *P'* will be *deny* effect. The result of the *RCA* depends on the result of evaluation of other rules including that of ith rule. Table 3.3 presents the possible cases for the evaluation of other rules (in the second column), along with the evaluation of ith rule in *P* and *P'* (in third and fourth column respectively) and result of *RCA* for both *P* and *P'* (in fifth and sixth column respectively). As shown in Table 3.3, there is the possibility of four possible evaluation of other rules.

The first possible evaluation is that - all other rules evaluate to *N/A* and produces no effect. In this case, the effect of ith rule in *P* - which is *permit* - will be the *RCA* level decision for *P* and that of *P'* will be *deny* since its ith rule has *deny* decision. When only one rule evaluates to *permit*, the result of *permit-overrides* RCA is *permit*. Similarly, when only one rule evaluates to *deny*, the result of *permit-overrides* RCA is *deny*. As a result, the RCA level decision is *permit* and *deny* in *P* and *P'* respectively as shown in the *fifth* and the *sixth* column of the first row respectively. The policy level decision depends on the result of RCA and how policy target evaluates as discussed in Section 2.2.2 and Section 2.2.3. When *PT* evaluates to *true* the policy-level result of *P* and *P'* will be the result of corresponding RCA i.e *permit* in P and deny in *P'*. Similarly, when *PT* evaluates to an error, the result of P and P' will be the Indeterminate of the result of RCA i.e the policy-level result of *P* is *I(P)* and that of *P'* is *I(D).*

Table 3.3: Possible Evaluations Of Other Rules when *Rb_i* Evaluates to True

	other rules than i^{th} rule	**i^{th} rule in P**	**i^{th} rule in P'**	**RCA in P**	**RCA in P'**
Produces effect	all other rules evaluate to *N/A* and produces no effect.	Permit	Deny	Permit	Deny
	One or more rule produces deny or I(D) effect and rest are *N/A*	Permit	Deny	Permit	Deny
	One or more rule produces I(P) effect and rest are *N/A*	Permit	Deny	Permit	I(DP)
	One or more rule produces I(P), one or more rule produces deny or I(D) effect and rest are *N/A*	Permit	Deny	Permit	I(DP)

Note: There is the possibility of another permit rule to be true but when it happens we could not detect the fault and the mutant will be equivalent to policy for such set of requests which makes another permit rule evaluates to true. Hence, for simplicity, we do not consider such cases in this table as well as in another table like this.

The second possible evaluation is when one or more deny rule produces a *deny* or *I(D)* effect and the rest are *N/A*. In this case, since the RCA is *permit-overrides*, the result of *RCA* in *P* is *permit* because i^{th} rule has *permit* effect and if any rule evaluates to permit, the result of *permit-overrides* is permit. In *P', i^{th}* rule evaluated to deny, one or more rules evaluated to *deny* or *I(D)* and other rules evaluated to *N/A* i.e none of the rules evaluated to *permit* or *I(P)*. When RCA is *permit-overrides*, if none of the rules evaluate to *permit* or *I(P)* and any one of the rules evaluate to *deny*, the RCA level decision will be *deny*. As a

result, the RCA level decision of *P'* is *deny*. When *PT* evaluates to *true* the result of P' will be *deny* and that of *P* is permit. Similarly, when *PT* evaluates to an error the result of *P'* is *I(D)* and that of *P* is *I(P)* which are different.

The third possible evaluation is when one or more permit rules produces *I(P)* effect and the rest are *N/A*. In this case, the result of *RCA* in *P* is *permit* since i^{th} rule has *permit* effect. For *P'*, i^{th} rule evaluates to deny and since one or more rules evaluated to *I(P)*, the result of RCA will be *I(DP)*. As a result, when *PT* evaluates to *true* the result of *P'* will be *I(DP)* and that of *P* is *permit* which is different. Similarly, when *PT* evaluates to an error the result of *P'* is *I(DP)* and that of *P* is *I(P)* which are different.

The fourth possible evaluation is one or more rules produces *I(P)* effect, one or more rules produces *deny* or *I(D)* effect and the rest are *N/A*. In this case, the result of *RCA* in *P* is still *permit* since i^{th} rule has *permit* effect and that of *P'* will be *I(DP)*. As a result, when *PT* evaluates to *true* the result of *P'* will be *I(DP)* and that of P is permit which is different. Similarly, when *PT* evaluates to an error the result of *P'* is *I(DP)* and that of *P* is *I(P)* which are different.

Hence, *when* rb_i *evaluates to true*, Þ is sufficient for propagation as it holds for all possible evaluation of other rules.

When rb_i evaluates to an error, Þ is not sufficient for propagation. Table 3.4 presents the rationale why Þ is not sufficient for propagation when rb_i evaluates to an error.

With similar reasoning as for Table 3.3, we can evaluate the result for *RCA* in *P* and *P'* as shown in Table 3.4. As listed in Table 3.4, the result of *RCA* in *P* and *P'* for the first three possible cases are different. Hence for them, Þ is sufficient for propagation i.e fault

detection. However, for the fourth case, the result of *RCA* in both *P* and *P'* is *I(DP)*. Since the result of *RCA* is not different and hence the policy level decision will not differ.

Table 3.4: Possible Evaluations of Other Rules when Rb_i Evaluates to Error

	other rules than ith rule	**ith rule in P**	**ith rule in P'**	**RCA in P**	**RCA in P'**
Produces effect	No effect from other rules as all of them evaluates to *N/A*	I(P)	I(D)	I(P)	I(D)
	One or more rule produces deny or I(D) effect and rest are *N/A*	I(P)	I(D)	Permit /I(P)	I(DP)
	One or more rule produces I(P) effect and rest are *N/A*	I(P)	I(D)	I(DP)	I(D)
	One or more rule produces I(P), one or more rule produces deny or I(D) effect and rest are *N/A*	I(P)	I(D)	I(DP)	I(DP)

Hence, Þ is insufficient for propagation for the fourth case when rb_i evaluates to an error. As a result, when rb_i evaluates to an error, in addition to Þ, we need additional constraint in propagation constraint. The required additional constraint is - there should not exist a pair of the rules (excluding i^{th} rule) such that one of them has *permit* effect which evaluates to an error and other has *deny* effect which evaluates to *true* or *error*. If this constraint is satisfied, then the fourth case in Table 3.4 will never occur and we can distinguish the fault and if this constraint is violated the fourth case will occur in which case we cannot distinguish the faulty policy.

Hence, the propagation constraint when rb_i evaluates to an error is that - any *permit* rule r_j $(j>i)$ after r_i should not evaluate to a *permit* decision and there should not exist a pair of the rules (excluding i^{th} rule) such that one of them has *permit* effect which evaluates to an error and other has *deny* effect which evaluates to *true* or *error*. Formally, additional constraint to Þ can be specified as $\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_p = < rb_p, permit>) \wedge (r_d = < rb_d, deny>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))$.

Combining all constraints as a single constraint, we get the following constraint for fault detection of CRE when RCA is *permit-overrides*.

$(\{PT \vee Error(PT)\} \wedge \{ rb_i\} \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \mathrel{!}= i\})$

$\vee$

$(\{PT \vee Error(PT)\} \wedge \{Error(rb_i)\} \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \mathrel{!}= i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_p = < rb_p, permit>) \wedge (r_d = < rb_d, deny>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

b) Deny-overrides:

i) Reachability constraint

The policy target should be *true or* should evaluate to an error. Further, when the rule-combining algorithm is *deny-overrides*, rule r_i will not be triggered if there is a *deny* rule before rule r_i that evaluates to a *deny* decision. Thus, the reachability constraint is that for any *deny* rule r_j $(j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with deny effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any deny rule* $r_j = \langle rb_j, deny\rangle$ *for* $j < i\}$

ii) <u>*Necessity constraint*</u>

rb_i should evaluate to either *true* so that the ith rule decision will be *permit* in *P* and *deny* in *P'* respectively or should *evaluate to an error* so that rule decision will be *I(P)* in *P* and *I(D) in P'*. If rb_i evaluates to *N/A*, the rule level decisions will be *N/A* in both *P* and *P'* in which case we can not distinguish the faulty policy. Hence, formally, necessity constraint is $rb_i \vee Error(rb_i)$.

iii) <u>*Propagation constraint*</u>

Any *deny* rule r_j $(j>i)$ after r_i, r_j should not evaluate to a *permit* decision, otherwise, $d(P, q) = d(P', q) = Deny$. The test must make rb_j evaluate to *N/A* or *error* for each *deny* rule r_j $(j>i)$. Therefore, the *propagation constraint* can be formalized as $\neg rb_j \vee Error(rb_j)$ for any rule $r_j = \langle rb_j, Deny\rangle$ $(j>i)$.

The afore-mentioned propagation constraint is sufficient for propagation if necessity constraint is rb_i evaluates to *true* but is not sufficient when rb_i evaluates to an error. With similar reasoning as for *permit-overrides*, we can state that when rb_i evaluates to an error, we need additional constraint for propagation. The required additional constraint is "any *deny* rule r_j $(j>i)$ after r_i should not evaluate to a *deny* decision and there should not exist a pair of the rules (excluding i^{th} rule) such that one of them has *deny* effect which evaluates to an error and other has *permit* effect which evaluates to *true* or *error*". Formally,

additional constraint is $\neg(\exists(p,d)$ *such that* $(i \neq p \neq d) \wedge (r_p = < rb_p, permit>) \wedge (r_d = < rb_d, deny>) \wedge Error(rb_d) \wedge (Error(rb_p) \vee rb_p))$.

Combining all constraints as a single constraint, we get the following constraint for fault detection of CRE when RCA is *deny-overrides*.

$(\{PT \vee Error(PT)\} \wedge \{rb_i\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any deny rule* $r_j = <rb_j, deny>$ *for* $j \mathrel{!}= i\})$

$\vee$

$(\{PT \vee Error(PT)\} \wedge \{Error(rb_i)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, deny>$ *for* $j \mathrel{!}= i\} \wedge \{\neg(\exists(p,d)$ *such that* $(i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_d) \wedge (Error(rb_p) \vee rb_p))\})$

Note: The fault detection condition for permit-overrides is symmetrical with fault detection condition for deny-overrides such that the role of the permit and deny effect are interchanged. Hence, for simplicity, we would only consider permit-overrides onwards for other faults.

c) ***deny-unless-permit*:**

i) *Reachability constraint*

The policy target should be *true or* should evaluate to an error. When the RCA is *deny-unless-permit*, rule r_i will not be triggered if there is a *permit* rule before rule r_i that evaluates to a *permit* decision. Thus, the reachability constraint is that for any *permit* rule $r_j = <rb_j, permit>$ $(j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j < i\}$.

ii) <u>*Necessity constraint*</u>

deny-unless-permit results *permit* decision only if at least one of the rules with *permit* effect evaluates to *true*, otherwise, it is *deny* in all other cases. As a result, we could not consider the *error* condition in the ith rule because if there is *error* in the ith rule target and/or condition, it will contribute to *deny* decision for the RCA in both P and P' irrespective of its effect. Hence, both P and P' will behave similarly if the current rule under consideration evaluates to an error. As a result, the necessity constraint should be the only rb_i evaluates to *true*. Formally, necessity constraint is rb_i.

iii) <u>*Propagation constraint*</u>

Any *permit* rule r_j $(j>i)$ after r_i, r_j should not evaluate to a *permit* decision, otherwise, $d(P, q)= d(P', q) = Permit$. The test must make rb_j evaluate to *N/A* or *error* for each *permit* rule r_j $(j>i)$. Therefore, the *propagation constraint* can be formalized as $\neg rb_j \vee Error(rb_j)$ for any rule $r_j = <rb_j, Permit>$ $(j>i)$.

Combining all constraints as a single constraint, we get the following constraint for fault detection of CRE when RCA is *deny-unless-permit*.

$(\{PT \vee Error(PT)\} \wedge \{ rb_i\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = < rb_j, Permit >$ *for* $j != i\})$

d) *permit-unless-deny*:

i) <u>*Reachability constraint*</u>

When the RCA is *permit-unless-deny*, rule r_i will not be triggered if there is a *deny* rule before rule r_i that evaluates to a *permit* decision. Thus, the reachability constraint is

that for any *deny* rule r_j $(j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with deny effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any deny rule* $r_j = \langle rb_j, deny\rangle$ *for* $j < i\}$

ii) <u>*Necessity constraint*</u>

permit-unless-deny results *deny* only if at least one of the rules with *deny* effect evaluates to *true*, otherwise, it is *permit* in all other cases. As a result, we could not consider the *error* condition in the ith rule because if there is *error* in the ith rule target and/or condition, it will contribute to *permit* decision for the RCA in both P and P' irrespective of its effect. As a result, the necessity constraint should be the only rb_i evaluates to *true*. Hence, formally, necessity constraint is rb_i.

iii) <u>*Propagation constraint*</u>

Any *deny* rule r_j $(j>i)$ after r_i, r_j should not evaluate to a *deny* decision, otherwise, $d(P, q)= d(P', q) = Deny$. The test must make rb_j evaluate to *N/A* or *error* for each *deny* rule r_j $(j>i)$. Therefore, the *propagation constraint* can be formalized as $\neg rb_j \vee Error(rb_j)$ for any rule $r_j = \langle rb_j, Deny\rangle$ $(j>i)$.

Combining all constraints as a single constraint, we get the following constraint for fault detection of CRE when RCA is *permit-unless-deny*.

$(\{PT \vee Error(PT)\} \wedge \{ rb_i\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = \langle rb_j, Deny\rangle$ *for* j $!= i\})$

Note: The fault detection condition for deny-unless-permit is symmetrical with fault detection condition for permit-unless-deny such that the role of the permit and deny effect are interchanged. Hence, for simplicity, we would only consider deny-unless-permit for other faults.

e) *first-applicable*:

i) *Reachability constraint*

When the rule-combining algorithm is *first-applicable*, rule r_i will not be triggered if there is any rule before rule r_i that evaluates to *true* or *error*. Thus, the reachability constraint is that - the policy target is *true or* evaluates to an error and for any rule r_j $(j<i)$ before rule r_i, rb_j should evaluate to *N/A*.

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j)$ *for any rule* r_j *for* $j < i\}$

ii) *Necessity constraint*

rb_i should evaluate to either *true* so that the ith rule decision will be *permit* in *P* and *deny* in *P'* respectively or should *evaluate to an error* so that rule decision will be *I(P)* in *P* and *I(D) in P'*. If rb_i evaluates to *N/A*, the rule level decisions will be *N/A* in both *P* and *P'* in which case we cannot distinguish the faulty policy. Hence, formally, necessity constraint is $rb_i \vee Error(rb_i)$.

iii) *Propagation constraint*

First-applicable RCA for CRE does not require explicit propagation constraint because reachability and necessity constraint is enough for fault detection.

Combining all constraints as a single constraint, we get the following constraint for fault detection of CRE when RCA is *first-applicable*.

$\{PT \vee Error(PT)\} \wedge \{rb_i \vee Error(rb_i)\} \wedge \{\neg rb_j)$ *for any rule* r_j *for* $j < i\}$

3.2 FDC for Rule Target True (RTT)

Rule target *true* is a mutation operator which alters the rule target such that it will always evaluate to *true*. One of the transformation rules to make target always evaluate to *true* is to make it empty so that it will always evaluate to *true*. Since it has a fault in the target of the rule, it is under the category incorrect rule target.

Consider an XACML policy $P = <PT, RCA, RL>$ and $P' = <PT, RCA, RL'>$ where RL = $<r_1, \ldots, r_i, \ldots, r_n>$, $RL' = <r_1, \ldots, r_i', \ldots, r_n>$, $r_i = <rt_i, rc_i, re_i>$ and $r_i' = <rt_i', rc_i, re_i>$ such that P' is similar to P except the target rt_i' of ith rule r_i' in P' always evaluates to *true*. Here, P' is called the Rule Target True (RTT) Mutant of P. The fault detection condition for Rule Target True based on RCA are given below.

a) Permit-overrides:

i) Reachability constraint

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j < i\}$

ii) Necessity constraint

The necessity constraint is that – rule target evaluates to *N/A* or *error* and rule condition evaluates to *true*. Formally, $(\neg rt_i \vee Error(rt_i)) \wedge rc_i$

iii) Propagation constraint

If policy target evaluates to *true*, i^{th} rule target evaluates to *N/A* and i^{th} rule effect is *deny*, then all other rules should evaluate to *N/A* or *error*. Further, there should not exist a

pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$(\{PT)\} \wedge \neg rt_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j)$ *for any rule* r_j *for* $j \;!= i\} \wedge \{\neg(\exists(p,d)$ *such that* $(i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

If policy target evaluates to an error, i^{th} rule target evaluates to *N/A* and i^{th} rule effect is *deny*, then all other rules with *deny* effect should evaluate to *N/A* and all other rules with *permit* effect should not evaluate to *true*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$(\{Error(PT)\} \wedge \neg rt_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_i, permit>$ *for* $j \;!= i\} \wedge \{\neg rb_j$ *for any deny rule* $r_j = <rb_i, deny>$ *for* $j \;!= i\} \wedge \{\neg(\exists(p,d)$ *such that* $(i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

If policy target evaluates to *true*, i^{th} rule target evaluates to *N/A* and i^{th} rule effect is *permit*, then all other permit rules should evaluate to *N/A* or *error*.

$(\{PT\} \wedge \neg rt_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_i, permit>$ *for* $j \;!= i\})$

If policy target evaluates to an error, i^{th} rule target evaluates to *N/A* and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A* if all *deny* rules evaluate to *N/A*. However, if any *deny* rule evaluates to *true* or *error*, then another permit effect can evaluate to *N/A* or *error*.

$$(\{Error(PT)\} \land \neg rt_i \land r_i = <rb_i, permit> \land [\{\neg rb_j \text{ for any rule } r_j \text{ for } j \,!= i\} \lor \{ \exists d \,(rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j)) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \,!= i \}])$$

If policy target evaluates to *true*, i^{th} rule target evaluates to an error and i^{th} rule effect is *deny*, then all other rules with *permit* effect should evaluate to *N/A* and all other rules should with *deny* effect should evaluate to *N/A* or *error*.

$$(\{PT)\} \land Error(rt_i) \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j) \text{ for any deny rule } r_j = <rb_j, deny> \text{ for } j \,!= i\} \land \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \,!= i\}$$

If policy target evaluates to an error, i^{th} rule target evaluates to an error and i^{th} rule effect is *deny*, then we cannot detect the fault and mutant will behave the same as original policy for these set of requests.

If policy target evaluates to *true*, i^{th} rule target evaluates to an error and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A*.

$$(\{PT)\} \land Error(rt_i) \land r_i = <rb_i, permit> \land \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \,!= i\})$$

If policy target evaluates to an error, i^{th} rule target evaluates to an error and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A* if all other deny rules evaluatesto an error. However, if any one deny rule evaluates to *true* or *error*, another permit rule can evaluate to an error or *N/A*.

$$(\{Error(PT)\} \land Error(rt_i) \land r_i = <rb_i, permit> \land [\{\neg rb_j \text{ for any rule } r_j \text{ for } j \,!= i\} \lor \{\exists d \,(rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j)) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \,!= i\}])$$

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is *permit-overrides*:

$$(\{PT\} \wedge \neg rt_i \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j \text{ for } j != i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

$\vee$

$$(\{Error(PT)\} \wedge \neg rt_i \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j != i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j != i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

$\vee$

$$(\{PT\} \wedge \neg rt_i \wedge rc_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j != i\})$$

$\vee$

$$(\{Error(PT)\} \wedge \neg rt_i \wedge rc_i \wedge r_i = <rb_i, permit> \wedge [\{\neg rb_j \text{ for any rule } r_j \text{ for } j != i\} \vee \{\exists d\ (rb_d \vee Error(rb_d)) \wedge (r_d = <rb_d, deny>) \wedge (\neg rb_j \vee Error(rb_j)) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j != i\}])$$

$\vee$

$$(\{PT)\} \wedge Error(rt_i) \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any deny rule } r_j = <rb_j, deny> \text{ for } j != i\} \wedge \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j != i\}$$

$\vee$

$$(\{PT)\} \wedge Error(rt_i) \wedge rc_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j != i\})$$

$\vee$

$(\{Error(PT)\} \wedge Error(rt_i) \wedge rc_i \wedge r_i = <rb_i, permit> \wedge [\{\neg rb_j \text{ for any rule } r_j \text{ for } j != i\} \vee \{\exists d\ (rb_d \vee Error(rb_d)) \wedge (r_d = <rb_d, deny>) \wedge (\neg rb_j \vee Error(rb_j)) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j != i\}])$

We can verify the validity of this fault detection condition by adopting similar reasoning that we applied for CRE. Let i^{th} rule have *deny* effect. The result of the RCA depends on the result of evaluation of other rules as well as that of ith rule. Table 3.5 presents the possible cases for evaluation of other rules and result of RCA for both *P* and *P'*.

With similar reasoning as for Table 3.3, we can list the records in Table 3.5. The final case in Table 3.5, when one or more rules (excluding i^{th} rule) produces *I(P)* effect and one or more other rules produces deny effect and rest of the rules are *N/A*, the result of RCA is similar in both P' and P (which implies that we cannot detect fault under those conditions). Additionally, in the second case - when one or more of the other rules evaluate to *deny* and rest as *N/A*, the result of RCA in P and P' are same. Hence from Table 3.5, we can conclude that if the effect of ith rule is *deny* then for fault detection, all the rules except the ith rule should not be *true* and there should not exists a pair of the rules (excluding i^{th} rule) such that one of them has *deny* effect, another has *permit* effect, *permit* rule evaluates to an error, and *deny* rule evaluates to *true* or *error*.

Table 3.5: Possible Evaluations of Other Rules When Rt_i Evaluates to *N/A* and R_i Is *Deny* Rule in P

	other rules than i^{th} rule	**i^{th} rule in P**	**i^{th} rule in P'**	**RCA in P**	**RCA in P'**
Produces effect	No effect from other rules as all of them evaluates to *N/A*	N/A	deny	N/A	deny
	One or more rule produces *deny* effect and rest are *N/A*	N/A	deny	deny	deny
	One or more rule produces *I(D)* effect and rest are *N/A*	N/A	deny	I(D)	deny
	One or more rule produces *I(P)* effect and rest are *N/A*	N/A	deny	I(P)	I(DP)
	One or more rule produces *I(P)*, one or more rule produces *deny* or *I(D)* effect and rest are *N/A*	N/A	deny	I(DP)	I(DP)

Further, if *PT* evaluates to an error then the result of both P and P' will be *I(D)* for the third case when one or more other rules with *deny* effect evaluate to *I(D).* Hence, when PT evaluates to an error, the fault detection condition is such that other rules with *permit* effect can evaluate to an error while that with *deny* effect should evaluate to *N/A*.

Now consider the effect of ith rule is *permit* instead of *deny*. In such case, Table 3.6 presents the possible cases for evaluation of other rules and result of RCA for both P and P'.

Table 3.6. Possible Evaluations of Other Rules When Rt_i Evaluates to *N/A* And R_i is *Permit* Rule in P

	other rules than i^{th} rule	**i^{th} rule in P**	**i^{th} rule in P'**	**RCA in P**	**RCA in P'**
Produces effect	No effect from other rules as all of them evaluates to *N/A*	N/A	permit	N/A	permit
	One or more rule produces *deny* effect and rest are *N/A*	N/A	permit	deny	permit
	One or more rule produces *I(D)* effect and rest are *N/A*	N/A	permit	I(D)	permit
	One or more rule produces *I(P)* effect and rest are *N/A*	N/A	permit	I(P)	permit
	One or more rule produces *I(P)*, one or more rule produces deny or *I(D)* effect and rest are *N/A*	N/A	permit	I(DP)	permit

It is evident from the cases in Table 3.6 that when the effect of the ith rule is *permit*, it can identify the fault if none of the other permit rule evaluates to *true*.

Further, if *PT* evaluates to an error then result of both P and P' will be *I(P)* for the fourth case when the rule with the *permit* effect evaluates to an error. Hence, when *PT* evaluates to an error, the fault detection condition is such that rules with the *permit* effect can only evaluate to *N/A*. However, if there is an another deny rule which does not evaluate to *N/A* then there could be a *permit* rule which evaluates to an error which is nothing but the final case in which case we can distinguish the fault.

Now, let's consider the situation when i^{th} rule target evaluates to an error. Table 3.7 presents the possible cases for evaluation of other rules and result of RCA for both *P* and *P'*. From Table 3.7, it is evident that fault can be detected only when other rules with *permit* effect evaluate to *N/A* and rules with *deny* effect do not evaluate to *true*.

Table 3.7: Possible Evaluations of Other Rules When Rt_i Evaluates to *Error* And R_i is *Deny* Rule in *P*

	other rules than i^{th} rule	**i^{th} rule in P**	**i^{th} rule in P'**	**RCA in P**	**RCA in P'**
Produces effect	No effect from other rules as all of them evaluates to *N/A*	I(D)	deny	I(D)	deny
	One or more rule produces *deny* effect and rest are *N/A*	I(D)	deny	deny	deny
	One or more rule produces *I(D)* effect and rest are *N/A*	I(D)	deny	I(D)	deny
	One or more rule produces *I(P)* effect and rest are *N/A*	I(D)	deny	I(DP)	I(DP)
	One or more rule produces *I(P)*, one or more rule produces *deny* or *I(P)* effect and rest are *N/A*	I(D)	deny	I(DP)	I(DP)

Further, if *PT* evaluates to an error then result of both P and P' will be the same in all cases and hence mutant behaves similarly as original policy.

Now consider the effect of ith rule is *permit* instead of *deny*. In such case, Table 3.8 presents the possible cases for evaluation of other rules and result of RCA for both P and P'.

Table 3.8. Possible Evaluations of Other Rules when Rt_i Evaluates to Error and R_i is *Permit* Rule in P

	other rules than i^{th} rule	i^{th} rule in P	i^{th} rule in P'	RCA in P	RCA in P'
Produces effect	No effect from other rules as all of them evaluates to *N/A*	I(P)	permit	I(P)	permit
	One or more rule produces *deny* effect and rest are *N/A*	I(P)	permit	I(DP)	permit
	One or more rule produces *I(D)* effect and rest are *N/A*	I(P)	permit	I(DP)	permit
	One or more rule produces *I(P)* effect and rest are *N/A*	I(P)	permit	I(P)	permit
	One or more rule produces *I(P)*, one or more rule produces *deny* or *I(D)* effect and rest are *N/A*	I(P)	permit	I(DP)	permit

It is evident from the above cases in Table 3.8 that when the effect of the ith rule is *permit*, it can identify the fault if all other permit rules does not evaluate to *true*.

Further, if *PT* evaluates to an error then the fault detection condition is such that other rules with *permit* effect cannot evaluate to *true* or *error* if all of the *deny* rules evaluate to

N/A. However, if at least one other deny rule evaluates to *true* or *error*, then another permit rule can evaluate to an error or *N/A*.

Note: We can establish the validity of the fault detection condition for all faults with similar reasoning as we did above for CRE and RTT. Hence, further in this work for simplicity, we only specify the fault detection condition precisely.

b) *deny-unless-permit*:

i) Reachability constraint

The policy target should be *true or* should evaluate to an error. For any *permit* rule $r_j = \langle rb_j, permit \rangle$ $(j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with *permit* effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = \langle rb_j, permit \rangle$ *for* $j < i\}$

ii) Necessity constraint

The target of the rule under consideration should evaluate to *N/A* or *error* and condition should evaluate to *true*. Formally, $(\neg rt_i \vee Error(rt_i)) \wedge rc_i$

iii) Propagation constraint

If rule under consideration is *deny* rule, then the mutant is equivalent because the rule level decision in mutant will be deny, and, in original policy, it will be N/A or error. However, for *deny-unless-permit* RCA, anything other than *permit* effect results in *deny* decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both of them will behave the same.

Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect after the ith rule should evaluate to *N/A* or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is *deny-unless-permit*.

$(\{PT \vee Error(PT)\} \wedge \{(\neg rt_i \vee Error(rt_i)) \wedge rc_i\} \wedge \{\neg rb_j \vee Error(rb_i)$ *for any permit rule* $r_j = < rb_j, permit >$ *for* $j != i\})$

***c) first-applicable*:**

i) *Reachability constraint*

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A*.

ii) *Necessity constraint*

If policy target evaluates to *true*, the necessity constraint is that – rule target evaluates to *N/A* or *error* and rule condition evaluates to *true*.

If policy target evaluates to an error, the necessity constraint is that – rule target evaluates to *N/A* and rule condition evaluates to *true*.

iii) *Propagation constraint*

If policy target evaluates to *true* and i^{th} rule target evaluates to *false*, then for all other rules which have the same effect as that of ith rule should evaluate to *N/A* or *error*.

If policy target evaluates to an error and i^{th} rule target evaluates to *false*, then for all other rules which have the same effect as that of ith rule should evaluate to *N/A*.

If policy target evaluates to *true* and i^{th} rule target evaluates to an error, then explicit propagation constraint is not required such that reachability and necessity constraint are enough for fault detection.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is *first-applicable*.

$(\{PT\} \wedge \neg rt_i \wedge rc_i \wedge \{\neg rb_j\ for\ any\ rule\ r_j\ for\ j < i\} \wedge \{\neg rb_j \vee Error(rb_j)\ for\ any\ rule\ r_j$ such that $re_i = re_j\ for\ j > i\})$

$\vee$

$(Error(PT)\} \wedge \neg rt_i \wedge rc_i \wedge \{\neg rb_j\ for\ any\ rule\ r_j\ for\ j < i\} \wedge \{\neg rb_j\ for\ any\ rule\ r_j$ such that $re_i = re_j\ for\ j > i\})$

$\vee$

$(PT \wedge Error(rt_i) \wedge rc_i \wedge \{\neg rb_j\ for\ any\ rule\ r_j\ for\ j < i\})$

3.3 FDC for Rule Target False (RTF)

Rule target *false* is a mutation operator which alters the rule target such that it will always evaluate to *false*. One of the transformation rules to make target always evaluate to *false* is to introduce new constraint in the target with a random attribute which will always be *false*. Since it has a fault in the target of the rule, it is under the category incorrect rule target.

Consider an XACML policy $P = <PT, RCA, RL>$ and $P' = <PT, RCA, RL'>$ where RL = $<r_1, ,\ldots, r_i, \ldots, r_n>$, $RL' = <r_1, \ldots, r_i', \ldots, r_n>$, $r_i = <rt_i, rc_i, re_i>$ and $r_i' = <rt_i', rc_i, re_i>$ such that P' is similar to P except the target rt_i' of ith rule r_i' in P' always evaluates to *false*. Here, P' is called the Rule Target False (RTF) Mutant of P. The fault detection condition for Rule Target False based on RCA are given below.

a) Permit-overrides:

i) Reachability constraint

Policy target should evaluate to *true* or *error* and all the previous rule with permit effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j < i\}$

ii) <u>*Necessity constraint*</u>

The necessity constraint is that – rule target evaluates to *true* or *error* and rule condition evaluates to *true*. Formally, $(rt_i \vee Error(rt_i)) \wedge rc_i$

iii) <u>*Propagation constraint*</u>

If policy target evaluates to *true*, i^{th} rule target evaluates to *true* and i^{th} rule effect is *deny*, then all other rules should evaluate to *N/A* or *error*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$(\{PT)\} \wedge rt_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j)$ *for any rule* r_j *for* $j \mathrel{!}= i\} \wedge \{\neg(\exists(p,d)$ *such that* $(i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

If policy target evaluates to an error, i^{th} rule target evaluates to *true* and i^{th} rule effect is *deny*, then all other rules with *deny* effect should evaluate to *N/A* and all other rules with *permit* effect should not evaluate to *true*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$$(\{Error(PT)\} \wedge rt_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \mathrel{!}= i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j \mathrel{!}= i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

If policy target evaluates to *true*, i^{th} rule target evaluates to *true* and i^{th} rule effect is *permit*, then all other permit rule should evaluate to *N/A* or *error*.

$$(\{PT\} \wedge rt_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \mathrel{!}= i\})$$

If policy target evaluates to an error, i^{th} rule target evaluates to *true* and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A* if all *deny* rules evaluate to *N/A*. However, if any *deny* rule evaluates to *true* or *error*, then another permit effect can evaluate to *N/A* or *error*.

$$(\{Error(PT)\} \wedge rt_i \wedge r_i = <rb_i, permit> \wedge [\{\neg rb_j \text{ for any rule } r_j \text{ for } j \mathrel{!}= i\} \vee \{ \exists d (rb_d \vee Error(rb_d)) \wedge (r_d = <rb_d, deny>) \wedge (\neg rb_j \vee Error(rb_j)) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \mathrel{!}= i \}])$$

If policy target evaluates to *true or error*, i^{th} rule target evaluates to an error and i^{th} rule effect is *deny*, then all other rules with *deny* effect should evaluate to *N/A* and all other rules with *permit* effect should evaluate to *N/A* or *error*.

$$(\{PT \vee Error(PT)\} \wedge Error(rt_i) \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \mathrel{!}= i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_j, deny> \text{ for } j \mathrel{!}= i\}$$

If policy target evaluates to *true or error*, i^{th} rule target evaluates to an error and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A*.

$(\{PT \vee Error(PT)\} \wedge Error(rt_i) \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \text{ != } i\})$

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTF when RCA is *permit-overrides:*

$$(\{PT\} \wedge rt_i \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j \text{ for } j \text{ != } i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = <rb_d, deny>) \wedge (r_p = <rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

$\vee$

$$(\{Error(PT)\} \wedge rt_i \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \text{ != } i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j \text{ != } i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = <rb_d, deny>) \wedge (r_p = <rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

$\vee$

$$(\{PT\} \wedge rt_i \wedge rc_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \text{ != } i\})$$

$\vee$

$$(\{Error(PT)\} \wedge rt_i \wedge rc_i \wedge r_i = <rb_i, permit> \wedge [\{\neg rb_j \text{ for any rule } r_j \text{ for } j \text{ != } i\} \vee \{\exists d\,(rb_d \vee Error(rb_d)) \wedge (r_d = <rb_d, deny>) \wedge (\neg rb_j \vee Error(rb_j)) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \text{ != } i\}])$$

$\vee$

$$(\{PT \vee Error(PT)\} \wedge Error(rt_i) \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \text{ != } i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_j, deny> \text{ for } j \text{ != } i\}$$

$\vee$

$(\{PT \vee Error(PT))\} \wedge Error(rt_i) \wedge rc_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j != i\})$

b) *deny-unless-permit*:

i) <u>*Reachability constraint*</u>

The policy target should be *true or* should evaluate to an error. For any *permit* rule $r_j = <rb_j, permit>$ $(j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with *permit* effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j < i\}$

ii) <u>*Necessity constraint*</u>

The target of the rule under consideration should evaluate to *true* and condition should evaluate to *true*. Formally, $rt_i \wedge rc_i$.

iii) <u>*Propagation constraint*</u>

If rule under consideration is *deny* rule, then the mutant is equivalent because the rule level decision in mutant will be *N/A,* and, in original policy, it will be *true* or *error*. However, for *deny-unless-permit* RCA, anything other than *permit* effect results in *deny* decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both will behave the same.

Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect evaluate to *N/A* or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTF when RCA is *deny-unless-permit*.

$(\{PT \vee Error(PT)\} \wedge \{(rt_i) \wedge rc_i\} \wedge \{\neg rb_j \vee Error(rb_i)$ *for any permit rule* $r_j = < rb_j, permit > for\ j\ != i\})$

c) *first-applicable*:

i) <u>*Reachability constraint*</u>

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A*.

ii) <u>*Necessity constraint*</u>

If policy target evaluates to *true*, the necessity constraint is that – rule target evaluates to *N/A* or *error* and rule condition evaluates to *true*.

If policy target evaluates to an error, the necessity constraint is that – rule target evaluates to *N/A* and rule condition evaluates to *true*.

iii) <u>*Propagation constraint*</u>

If policy target evaluates to *true* and i^{th} rule target evaluates to *true*, then for all other rules which have the same effect as that of ith rule should evaluate to *N/A* or *error*.

If policy target evaluates to *true* and i^{th} rule target evaluates to an error, then for all other rules after the ith rule which has the same effect as that of ith rule should evaluate to *N/A* or *true*

If policy target evaluates to an error and i^{th} rule target evaluates to an error *or true*, then for all other rules after the ith rule which has the same effect as that of ith rule should evaluate to *N/A*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTF when RCA is *first-applicable*.

$(\{PT\} \wedge rt_i \wedge rc_i \wedge \{\neg rb_j$ *for any rule* r_j *for* $j < i\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any rule* r_j such that $re_i = re_j$ *for* $j > i\})$

$\vee$

$(\{PT\} \wedge Error(rt_i) \wedge rc_i \wedge \{\neg rb_j \vee rb_j$ *for any rule* r_j *for* $j < i\} \wedge \{\neg rb_j$ *for any rule* r_j such that $re_i = re_j$ *for* $j > i\})$

$\vee$

$(\{Error(PT)\} \wedge (rt_i \vee Error(rt_i)) \wedge rc_i \wedge \{\neg rb_j$ *for any rule* r_j *for* $j < i\} \wedge \{\neg rb_j$ *for any rule* r_j such that $re_i = re_j$ *for* $j > i\})$

3.4 FDC for Rule Condition True (RCT)

Rule condition *true* is a mutation operator which alters the rule condition such that it will always evaluate to *true*. One of the transformation rules to make condition always evaluate to *true* is to make it empty so that it will always evaluate to *true*. Since it has a fault in the condition of the rule, it is under the category incorrect rule condition.

Consider an XACML policy $P = <PT, RCA, RL>$ and $P' = <PT, RCA, RL'>$ where RL = $<r_1, ..., r_i, ..., r_n>$, $RL' = <r_1, ..., r_i', ..., r_n>$, $r_i = <rt_i, rc_i, re_i>$ and $r_i' = <rt_i', rc_i, re_i>$ such that P' is similar to P except the condition rc_i' of ith rule r_i' in P' always evaluates to *true*. Here, P' is called the Rule Condition True (RCT) Mutant of P. The fault detection condition for Rule Condition True based on RCA are given below.

a) Permit-overrides:

i) Reachability constraint

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = \langle rb_j, permit \rangle$ *for* $j < i\}$

ii) <u>Necessity constraint</u>

The necessity constraint is that – rule condition evaluates to *N/A* or *error* and rule target evaluates to *true*.

Formally, $(\neg rc_i \vee Error(rc_i)) \wedge rt_i$

iii) <u>Propagation constraint</u>

If policy target evaluates to *true*, i^{th} rule condition evaluates to *N/A* and i^{th} rule effect is *deny*, then all other rules should evaluate to *N/A* or *error*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$(\{PT)\} \wedge \neg rc_i \wedge r_i = \langle rb_i, deny \rangle \wedge \{\neg rb_j \vee Error(rb_j)$ *for any rule* r_j *for* $j \mathrel{!}= i\} \wedge \{\neg(\exists(p,d)$ *such that* $(i \neq p \neq d) \wedge (r_d = \langle rb_d, deny \rangle) \wedge (r_p = \langle rb_p, permit \rangle) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

If policy target evaluates to an error, i^{th} rule condition evaluates to *N/A* and i^{th} rule effect is *deny*, then all other rules with *deny* effect should evaluate to *N/A* and all other rules with *permit* effect should not evaluate to *true*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$$(\{Error(PT)\} \wedge \neg rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j != i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j != i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

If policy target evaluates to *true*, i^{th} rule condition evaluates to *N/A* and i^{th} rule effect is *permit*, then all other permit rules should evaluate to *N/A* or *error*.

$$(\{PT\} \wedge \neg rc_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j != i\})$$

If policy target evaluates to an error, i^{th} rule condition evaluates to *N/A* and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A* if all *deny* rules evaluate to *N/A*. However, if any *deny* rule evaluates to *true* or *error*, then another permit effect can evaluate to *N/A* or *error*.

$$(\{Error(PT)\} \wedge \neg rc_i \wedge r_i = <rb_i, permit> \wedge [\{\neg rb_j \text{ for any rule } r_j \text{ for } j != i\} \vee \{ \exists d (rb_d \vee Error(rb_d)) \wedge (r_d = <rb_d, deny>) \wedge (\neg rb_j \vee Error(rb_j)) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j != i \}])$$

If policy target evaluates to *true*, i^{th} rule condition evaluates to an error and i^{th} rule effect is *deny*, then all other rules with *permit* effect should evaluate to *N/A* and all other rules should with *deny* effect should evaluate to *N/A* or *error*.

$$(\{PT)\} \wedge Error(rc_i) \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any deny rule } r_j = <rb_j, deny> \text{ for } j != i\} \wedge \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j != i\}$$

If policy target evaluates to an error, i^{th} rule target evaluates to an error and i^{th} rule effect is *deny*, then we cannot detect the fault and mutant will behave the same as original policy for these set of requests.

If policy target evaluates to *true*, i^{th} rule condition evaluates to an error and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A*.

$(\{PT)\} \wedge Error(rc_i) \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \mathrel{!=} i\})$

If policy target evaluates to an error, i^{th} rule condition evaluates to an error and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A* if all other deny rules evaluate to an error. However, if any one deny rule evaluates to *true* or *error*, another permit rule can evaluate to an error or *N/A*.

$(\{Error(PT)\} \wedge Error(rc_i) \wedge r_i = <rb_i, permit> \wedge [\{\neg rb_j \text{ for any rule } r_j \text{ for } j \mathrel{!=} i\} \vee \{\exists d\,(rb_d \vee Error(rb_d)) \wedge (r_d = <rb_d, deny>) \wedge (\neg rb_j \vee Error(rb_j)) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \mathrel{!=} i\}])$

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCT when RCA is *permit-overrides:*

$(\{PT\} \wedge \neg rc_i \wedge rt_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j \text{ for } j \mathrel{!=} i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

$\vee$

$(\{Error(PT)\} \wedge \neg rc_i \wedge rt_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \mathrel{!=} i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j \mathrel{!=} i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

$\vee$

$(\{PT\} \land \neg rc_i \land rt_i \land r_i = <rb_i, permit> \land \{\neg rb_j \lor Error(rb_j)\ for\ any\ permit\ rule\ r_j =$

$<rb_i, permit>\ for\ j\ != i\})$

$\lor$

$(\{Error(PT)\} \land \neg rc_i \land rt_i \land r_i = <rb_i, permit> \land [\{\neg rb_j\ for\ any\ rule\ r_j\ for\ j\ != i\} \lor \{\exists d$

$(rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j))\ for\ any\ permit\ rule\ r_j =$

$<rb_i, permit>\ for\ j\ != i\}])$

$\lor$

$(\{PT)\} \land Error(rc_i) \land rt_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j)\ for\ any\ deny\ rule\ r_j =$

$<rb_j, deny>\ for\ j\ != i\} \land \{\neg rb_j\ for\ any\ permit\ rule\ r_j = <rb_j, permit>\ for\ j\ != i\}$

$\lor$

$(\{PT)\} \land Error(rc_i) \land rt_i \land r_i = <rb_i, permit> \land \{\neg rb_j\ for\ any\ permit\ rule\ r_j = <rb_j, permit>$

$for\ j\ != i\})$

$\lor$

$(\{Error(PT)\} \land Error(rc_i) \land rt_i \land r_i = <rb_i, permit> \land [\{\neg rb_j\ for\ any\ rule\ r_j\ for\ j\ != i\} \lor$

$\{\exists d\ (rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j))\ for\ any\ permit\ rule\ r_j =$

$<rb_j, permit>\ for\ j\ != i\}])$

b) *deny-unless-permit*:

i) <u>Reachability constraint</u>

The policy target should be *true or* should evaluate to an error. For any *permit* rule r_j $= <rb_j, permit>$ $(j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rule with *permit* effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = \langle rb_j, permit\rangle$ *for* $j < i$

ii) <u>*Necessity constraint*</u>

The condition of the rule under consideration should evaluate to *N/A* or *error* and target should evaluate to *true*. Formally, $(\neg rc_i \vee Error(rc_i)) \wedge rt_i$

iii) <u>*Propagation constraint*</u>

If rule under consideration is *deny* rule, then the mutant is equivalent because the rule level decision in mutant will be deny, and, in original policy, it will be N/A or error. However, for *deny-unless-permit* RCA, anything other than *permit* effect results in *deny* decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both of them will behave the same.

Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect after the ith rule should evaluate to *N/A* or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCT when RCA is *deny-unless-permit*.

$(\{PT \vee Error(PT)\} \wedge \{(\neg rc_i \vee Error(rc_i)) \wedge rt_i\} \wedge \{\neg rb_j \vee Error(rb_i)$ *for any permit rule* $r_j = \langle rb_j, permit\rangle$ *for* $j \,!\!= i\})$

c) ***first-applicable*:**

i) <u>*Reachability constraint*</u>

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A*.

ii) <u>*Necessity constraint*</u>

If policy target evaluates to *true*, the necessity constraint is that – rule condition evaluates to *N/A* or *error* and rule target evaluates to *true*.

If policy target evaluates to an error, the necessity constraint is that – rule condition evaluates to *N/A* and rule target evaluates to *true*.

iii) <u>*Propagation constraint*</u>

If policy target evaluates to *true* and i^{th} rule condition evaluates to *false*, then for all other rules which have the same effect as that of ith rule should evaluate to *N/A* or *error*.

If policy target evaluates to an error and i^{th} rule condition evaluates to *false*, then for all other rules which have the same effect as that of ith rule should evaluate to *N/A*.

If policy target evaluates to *true* and i^{th} rule condition evaluates to an error, then explicit propagation constraint is not required such that reachability and necessity constraint are enough for fault detection.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCT when RCA is *first-applicable*.

$(\{PT\} \wedge \neg rc_i \wedge rt_i \wedge \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j$ such that $re_i = re_j \text{ for } j > i\})$

$\vee$

$(Error(PT)\} \wedge \neg rc_i \wedge rt_i \wedge \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \wedge \{\neg rb_j \text{ for any rule } r_j$ such that $re_i = re_j \text{ for } j > i\})$

$\vee$

$(PT \wedge Error(rc_i) \wedge rt_i \wedge \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\})$

3.5 FDC for Rule Condition False (RCF)

Rule condition *false* is a mutation operator which alters the rule condition such that it will always evaluate to *false*. One of the transformation rules to make condition always evaluate to *false* is to introduce new constraint in condition with a random attribute which will always be *false*. Since it has a fault in the condition of the rule, it is under the category incorrect rule condition.

Consider an XACML policy $P = <PT, RCA, RL>$ and $P' = <PT, RCA, RL'>$ where RL = $<r_1, \ldots, r_i, \ldots, r_n>$, $RL' = <r_1, \ldots, r_i', \ldots, r_n>$, $r_i = <rt_i, rc_i, re_i>$ and $r_i' = <rt_i', rc_i, re_i>$ such that P' is similar to P except the condition rc_i' of ith rule r_i' in P' always evaluates to *false*. Here, P' is called the Rule Condition False (RCF) Mutant of P. The fault detection condition for Rule Condition False based on RCA are given below.

a) Permit-overrides:

i) Reachability constraint

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j < i\}$

ii) Necessity constraint

The necessity constraint is that – rule condition evaluates to *true* or *error* and rule target evaluates to *true*. Formally, $(rc_i \vee Error(rc_i)) \wedge rt_i$

iii) Propagation constraint

If policy target evaluates to *true*, i^{th} rule condition evaluates to *true* and i^{th} rule effect is *deny*, then all other rules should evaluate to *N/A* or *error*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$$(\{PT)\} \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j \text{ for } j \,!= i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

If policy target evaluates to an error, i^{th} rule condition evaluates to *true* and i^{th} rule effect is *deny*, then all other rules with *deny* effect should evaluate to *N/A* and all other rules with *permit* effect should not evaluate to *true*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$$(\{Error(PT)\} \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \,!= i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j \,!= i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

If policy target evaluates to *true*, i^{th} rule condition evaluates to *true* and i^{th} rule effect is *permit*, then all other permit rule should evaluate to *N/A* or *error*.

$$(\{PT\} \wedge rc_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \,!= i\})$$

If policy target evaluates to an error, i^{th} rule condition evaluates to *true* and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A* if all *deny*

rules evaluate to *N/A*. However, if any *deny* rule evaluates to *true* or *error*, then another permit effect can evaluate to *N/A* or *error*.

$(\{Error(PT)\} \wedge rc_i \wedge r_i = <rb_i, permit> \wedge [\{\neg rb_j \text{ for any rule } r_j \text{ for } j \,!= i\} \vee \{ \exists d\, (rb_d \vee Error(rb_d)) \wedge (r_d = <rb_d, deny>) \wedge (\neg rb_j \vee Error(rb_j)) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \,!= i \}])$

If policy target evaluates to *true or error*, i^{th} rule condition evaluates to an error and i^{th} rule effect is *deny*, then all other rules with *deny* effect should evaluate to *N/A* and all other rules with *permit* effect should evaluate to *N/A* or *error*.

$(\{PT \vee Error(PT)\} \wedge Error(rc_i) \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \,!= i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_j, deny> \text{ for } j \,!= i\}$

If policy target evaluates to *true or error*, i^{th} rule condition evaluates to an error and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A*.

$(\{PT \vee Error(PT)\} \wedge Error(rc_i) \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \,!= i\})$

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCF when RCA is *permit-overrides:*

$(\{PT\} \wedge rt_i \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j \text{ for } j \,!= i\} \wedge \{\neg(\exists (p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

$\vee$

$(\{Error(PT)\} \wedge rt_i \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \,!= i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j \,!= i\} \wedge \{\neg(\exists (p,d)$

such that $(i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

$\vee$

$(\{PT\} \wedge rt_i \wedge rc_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_i, permit>$ *for* $j != i\})$

$\vee$

$(\{Error(PT)\} \wedge rt_i \wedge rc_i \wedge r_i = <rb_i, permit> \wedge [\{\neg rb_j$ *for any rule* r_j *for* $j != i\} \vee \{ \exists d (rb_d \vee Error(rb_d)) \wedge (r_d = <rb_d, deny>) \wedge (\neg rb_j \vee Error(rb_j))$ *for any permit rule* $r_j = <rb_i, permit>$ *for* $j != i \}])$

$\vee$

$(\{PT \vee Error(PT)\} \wedge Error(rc_i) \wedge rt_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j != i\} \wedge \{\neg rb_j$ *for any deny rule* $r_j = <rb_j, deny>$ *for* $j != i\}$

$\vee$

$(\{PT \vee Error(PT))\} \wedge Error(rc_i) \wedge rt_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j != i\})$

b) *deny-unless-permit*:

<u>*i)* Reachability constraint</u>

The policy target should be *true or* should evaluate to an error. For any *permit* rule $r_j = <rb_j, permit> (j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with *permit* effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be *{PT* $\vee$ *Error(PT)}* $\wedge$ *{*$\neg rb_j \vee$ *Error*(rb_j) *for any permit rule* r_j = <rb_j, *permit*> *for* $j < i$*}*

ii) <u>*Necessity constraint*</u>

The target of the rule under consideration should evaluate to *true* and condition should evaluate to *true*. Formally, $rt_i \wedge rc_i$

iii) <u>*Propagation constraint*</u>

If rule under consideration is *deny* rule, then the mutant is equivalent because the rule level decision in mutant will be *N/A* and, in original policy, it will be *true* or *error. However,* for *deny-unless-permit* RCA, anything other than *permit* effect results in *deny* decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both will behave the same.

Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect evaluate to *N/A* or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCF when RCA is *deny-unless-permit*.

({PT $\vee$ *Error(PT)}* $\wedge$ *{*$rt_i \wedge rc_i$*}* $\wedge$ *{*$\neg rb_j \vee$ *Error*(rb_i) *for any permit rule* r_j = < rb_j, *permit* >*for* j != i*})*

c) ***first-applicable*:**

<u>*i)*</u> <u>*Reachability constraint*</u>

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A*.

ii) <u>*Necessity constraint*</u>

If policy target evaluates to *true*, the necessity constraint is that – rule condition evaluates to *N/A* or *error* and rule target evaluates to *true*.

If policy target evaluates to an error, the necessity constraint is that – rule condition evaluates to *N/A* and rule target evaluates to *true*.

iii) *Propagation constraint*

If policy target evaluates to *true* and i^{th} rule condition evaluates to *true*, then for all other rules which have the same effect as that of ith rule should evaluate to *N/A* or *error*.

If policy target evaluates to *true* and i^{th} rule condition evaluates to an error, then for all other rules after the ith rule which has the same effect as that of ith rule should evaluate to *N/A* or *true*

If policy target evaluates to an error and i^{th} rule condition evaluates to an error *or true*, then for all other rules after the ith rule which has the same effect as that of ith rule should evaluate to *N/A*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RCF when RCA is *first-applicable*.

$(\{PT\} \wedge rt_i \wedge rc_i \wedge \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j$ such that $re_i = re_j \text{ for } j > i\})$

$\vee$

$(\{PT\} \wedge Error(rc_i) \wedge rt_i \wedge \{\neg rb_j \vee rb_j \text{ for any rule } r_j \text{ for } j < i\} \wedge \{\neg rb_j \text{ for any rule } r_j$ such that $re_i = re_j \text{ for } j > i\})$

$\vee$

$(\{Error(PT)\} \wedge (rc_i \vee Error(rc_i)) \wedge rt_i \wedge \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \wedge \{\neg rb_j \text{ for any rule }$ r_j such that $re_i = re_j \text{ for } j > i\})$

3.6 FDC for Add Not Function (ANF)

Add not function is a mutation operator which adds a not function to the rule condition such that rule condition will always evaluate to *false* on faulty policy if it evaluates to *true* in original policy, and vice versa. Since it has a fault in the condition of the rule, it is under the category incorrect rule condition.

Consider an XACML policy $P = <PT, RCA, RL>$ and $P' = <PT, RCA, RL'>$ where RL = $<r_1, \ldots, r_i, \ldots, r_n>$, $RL' = <r_1, \ldots, r_i', \ldots, r_n>$, $r_i = <rt_i, rc_i, re_i>$ and $r_i' = <rt_i', rc_i, re_i>$ such that P' is similar to P except the condition rc_i' of ith rule r_i' in P' *is* $\neg rc_i$. Here, P' is called the Add Not Function (ANF) Mutant of P. The fault detection condition for ANF based on RCA are given below.

a) Permit-overrides:

i) Reachability constraint

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e. should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j < i\}$

ii) Necessity constraint

The necessity constraint is that – rule condition evaluates to *true* and rule target evaluates to *true*. Formally, $rc_i \wedge rt_i$

Note: we do not consider the error case on condition because error case will produce no different intermediate results between faulty and original policy

iii) Propagation constraint

If policy target evaluates to *true*, i^{th} rule condition evaluates to *true* and i^{th} rule effect is *deny*, then all other rules should evaluate to *N/A* or *error*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$$(\{PT)\} \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j \text{ for } j != i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

If policy target evaluates to an error, i^{th} rule condition evaluates to *true* and i^{th} rule effect is *deny*, then all other rules with *deny* effect should evaluate to *N/A* and all other rules with *permit* effect should not evaluate to *true*. Further, there should not exist a pair of the rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$$(\{Error(PT)\} \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j != i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j != i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

If policy target evaluates to *true*, i^{th} rule condition evaluates to *true* and i^{th} rule effect is *permit*, then all other permit rule should evaluate to *N/A* or *error*.

$$(\{PT\} \wedge rc_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j != i\})$$

If policy target evaluates to an error, i^{th} rule condition evaluates to *true* and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A* if all *deny*

rules evaluate to *N/A*. However, if any *deny* rule evaluates to *true* or *error*, then another permit effect can evaluate to *N/A* or *error*.

$$(\{Error(PT)\} \wedge rc_i \wedge r_i = \langle rb_i, permit\rangle \wedge [\{\neg rb_j \text{ for any rule } r_j \text{ for } j \mathrel{!=} i\} \vee \{\exists d\,(rb_d \vee Error(rb_d)) \wedge (r_d = \langle rb_d, deny\rangle) \wedge (\neg rb_j \vee Error(rb_j)) \text{ for any permit rule } r_j = \langle rb_i, permit\rangle \text{ for } j \mathrel{!=} i\}])$$

Combining all constraints as a single constraint, we get the following constraint for fault detection of ANF when RCA is *permit-overrides*:

$$(\{PT\} \wedge rt_i \wedge rc_i \wedge r_i = \langle rb_i, deny\rangle \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j \text{ for } j \mathrel{!=} i\} \wedge \{\neg(\exists (p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = \langle rb_d, deny\rangle) \wedge (r_p = \langle rb_p, permit\rangle) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

$$\vee$$

$$(\{Error(PT)\} \wedge rt_i \wedge rc_i \wedge r_i = \langle rb_i, deny\rangle \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = \langle rb_i, permit\rangle \text{ for } j \mathrel{!=} i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = \langle rb_i, deny\rangle \text{ for } j \mathrel{!=} i\} \wedge \{\neg(\exists (p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = \langle rb_d, deny\rangle) \wedge (r_p = \langle rb_p, permit\rangle) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$$

$$\vee$$

$$(\{PT\} \wedge rt_i \wedge rc_i \wedge r_i = \langle rb_i, permit\rangle \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = \langle rb_i, permit\rangle \text{ for } j \mathrel{!=} i\})$$

$$\vee$$

$$(\{Error(PT)\} \wedge rt_i \wedge rc_i \wedge r_i = \langle rb_i, permit\rangle \wedge [\{\neg rb_j \text{ for any rule } r_j \text{ for } j \mathrel{!=} i\} \vee \{\exists d\,(rb_d \vee Error(rb_d)) \wedge (r_d = \langle rb_d, deny\rangle) \wedge (\neg rb_j \vee Error(rb_j)) \text{ for any permit rule } r_j = \langle rb_i, permit\rangle \text{ for } j \mathrel{!=} i\}])$$

b) *deny-unless-permit*:

i) *Reachability constraint*

The policy target should be *true or* should evaluate to an error. For any *permit* rule $r_j = \langle rb_j, permit\rangle$ $(j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with *permit* effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = \langle rb_j, permit\rangle$ *for* $j < i\}$

ii) *Necessity constraint*

The target of the rule under consideration should evaluate to *true* and condition should evaluate to *true*. Formally, $rt_i \wedge rc_i$

iii) *Propagation constraint*

If rule under consideration is *deny* rule, then the mutant is equivalent because the rule level decision in mutant will be *N/A,* and, in original policy, it will be *true* or *error. However,* for *deny-unless-permit* RCA, anything other than *permit* effect results in *deny* decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both will behave the same.

Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect evaluate to *N/A* or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of ANF when RCA is *deny-unless-permit*.

$(\{PT \vee Error(PT)\} \wedge \{rt_i \wedge rc_i\} \wedge \{\neg rb_j \vee Error(rb_i)$ *for any permit rule* $r_j = < rb_j,$ *permit* $>$ *for* $j \,!= i\})$

c) ***first-applicable*:**

i) *Reachability constraint*

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A*.

ii) *Necessity constraint*

The necessity constraint is that – rule condition evaluates to *N/A* and rule target evaluates to *true*.

iii) *Propagation constraint*

If policy target evaluates to *true* and i^{th} rule condition evaluates to *true*, then for all other rules which have the same effect as that of ith rule should evaluate to *N/A* or *error*.

If policy target evaluates to an error and i^{th} rule condition evaluates *true*, then for all other rules after the ith rule which has the same effect as that of ith rule should evaluate to *N/A*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of ANF when RCA is *first-applicable*.

$(\{PT\} \wedge rt_i \wedge rc_i \wedge \{\neg rb_j$ *for any rule* r_j *for* $j < i\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any rule* r_j such that $re_i = re_j$ *for* $j > i\})$

$\vee$

$(\{Error(PT)\} \wedge rc_i \wedge rt_i \wedge \{\neg rb_j$ *for any rule* r_j *for* $j < i\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any rule* r_j such that $re_i = re_j$ *for* $j > i\})$

3.7 FDC for Remove Not Function (RNF)

Remove not function is a mutation operator which removes a not function from the rule condition such that rule condition will always evaluate to *false* on faulty policy if it evaluates to *true* in original policy, and vice versa. Since it has a fault in the condition of the rule, it is under the category incorrect rule condition.

Consider an XACML policy $P = <PT, RCA, RL>$ and $P' = <PT, RCA, RL'>$ where RL = $<r_1, \ldots ,r_i, \ldots, r_n>$, $RL' = <r_1, \ldots, r_i', \ldots, r_n>$, $r_i = <rt_i, rc_i, re_i>$ and $r_i' = <rt_i', rc_i, re_i>$ such that P' is similar to P except the condition rc_i' of ith rule r_i' in P' *is* $\neg rc_i$. Here, P' is called the Remove Not Function (RNF) Mutant of P. The fault detection condition for RNF is same as that for ANF because the effect of mutation operator on both cases is $rc_i' = \neg rc_i$.

3.8 FDC for Remove a Rule (RER)

Remove a rule is a mutation operator which removes a rule from the policy such that rule condition will always evaluate to *false* on faulty policy if it evaluates to *true* in original policy, and vice versa. Since it has a fault in the condition of the rule, it is under the category incorrect rule condition.

Consider an XACML policy $P = <PT, RCA, RL>$ and $P' = <PT, RCA, RL'>$ where RL = $<r_1, \ldots ,r_i, \ldots, r_n>$, $RL' = <r_1, \ldots, r_{i-1}, r_{i+1}, , \ldots, r_n>$, $r_i = <rt_i, rc_i, re_i>$ and $r_i' = <rt_i', rc_i, re_i>$ such that P' is similar to P except for that P' does not contain r_i in in rule list. Here, P' is called the Remove a Rule (RER) Mutant of P. The fault detection condition for RER based on RCA are given below.

a) Permit-overrides:

i) Reachability constraint

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j < i\}$

ii) *Necessity constraint*

The necessity constraint is that – rule body evaluates to *true* or *error*. Formally, $rb_i \vee Error(rb_i)$

iii) *Propagation constraint*

If policy target evaluates to *true*, i^{th} rule body evaluates to *true* and i^{th} rule effect is *deny*, then all other rules should evaluate to *N/A* or *error*. Further, there should not exist a pair of rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$(\{PT\}\} \wedge rb_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j)$ *for any rule* r_j *for* $j \,!= i\} \wedge \{\neg(\exists(p,d)$ *such that* $(i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

If policy target evaluates to an error, i^{th} rule body evaluates to *true* and i^{th} rule effect is *deny*, then all other rules with *deny* effect should evaluate to *N/A* and all other rules with *permit* effect should not evaluate to *true*. Further, there should not exist a pair of rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$(\{Error(PT)\} \wedge rb_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_i, permit>$ *for* $j \,!= i\} \wedge \{\neg rb_j$ *for any deny rule* $r_j = <rb_i, deny>$ *for* $j \,!= i\} \wedge \{\neg(\exists(p,d)$ *such*

that $(i \neq p \neq d) \wedge (r_d = \langle rb_d, deny \rangle) \wedge (r_p = \langle rb_p, permit \rangle) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

If policy target evaluates to *true*, i^{th} rule body evaluates to *true* and i^{th} rule effect is *permit*, then all other permit rule should evaluate to *N/A* or *error*.

$(\{PT\} \wedge rb_i \wedge r_i = \langle rb_i, permit \rangle \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = \langle rb_i, permit \rangle$ *for* $j \mathrel{!}= i\})$

If policy target evaluates to an error, i^{th} rule body evaluates to *true* and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A* if all *deny* rules evaluate to *N/A*. However, if any *deny* rule evaluates to *true* or *error*, then another permit effect can evaluate to *N/A* or *error*.

$(\{Error(PT)\} \wedge rb_i \wedge r_i = \langle rb_i, permit \rangle \wedge [\{\neg rb_j$ *for any rule* r_j *for* $j \mathrel{!}= i\} \vee \{\exists d\, (rb_d \vee Error(rb_d)) \wedge (r_d = \langle rb_d, deny \rangle) \wedge (\neg rb_j \vee Error(rb_j))$ *for any permit rule* $r_j = \langle rb_i, permit \rangle$ *for* $j \mathrel{!}= i\}])$

If policy target evaluates to *true or error*, i^{th} rule body evaluates to an error and i^{th} rule effect is *deny*, then all other rules with *deny* effect should evaluate to *N/A* and all other rules with *permit* effect should evaluate to *N/A* or *error*.

$(\{PT \vee Error(PT)\} \wedge Error(rb_i) \wedge r_i = \langle rb_i, deny \rangle \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = \langle rb_j, permit \rangle$ *for* $j \mathrel{!}= i\} \wedge \{\neg rb_j$ *for any deny rule* $r_j = \langle rb_j, deny \rangle$ *for* $j \mathrel{!}= i\}$

If policy target evaluates to *true or error*, i^{th} rule body evaluates to an error and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A*.

$(\{PT \vee Error(PT)\} \wedge Error(rb_i) \wedge r_i = \langle rb_i, permit \rangle \wedge \{\neg rb_j$ *for any permit rule* $r_j = \langle rb_j, permit \rangle$ *for* $j \mathrel{!}= i\})$

Combining all constraints as a single constraint, we get the following constraint for fault detection of RER when RCA is *permit-overrides:*

$(\{PT\} \land rb_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j) \text{ for any rule } r_j \text{ for } j \, != i\} \land \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \land (r_d = < rb_d, deny>) \land (r_p = < rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d))\})$

$\lor$

$(\{Error(PT)\} \land rb_i \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \, != i\} \land \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j \, != i\} \land \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \land (r_d = < rb_d, deny>) \land (r_p = < rb_p, permit>) \land Error(rb_p) \land (Error(rb_d) \lor rb_d))\})$

$\lor$

$(\{PT\} \land rb_i \land r_i = <rb_i, permit> \land \{\neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \, != i\})$

$\lor$

$(\{Error(PT)\} \land rb_i \land r_i = <rb_i, permit> \land [\{\neg rb_j \text{ for any rule } r_j \text{ for } j \, != i\} \lor \{ \exists d (rb_d \lor Error(rb_d)) \land (r_d = <rb_d, deny>) \land (\neg rb_j \lor Error(rb_j)) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \, != i \}])$

$\lor$

$(\{PT \lor Error(PT)\} \land Error(rb_i) \land r_i = <rb_i, deny> \land \{\neg rb_j \lor Error(rb_j) \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \, != i\} \land \{\neg rb_j \text{ for any deny rule } r_j = <rb_j, deny> \text{ for } j \, != i\}$

$\lor$

$(\{PT \lor Error(PT))\} \land Error(rb_i) \land r_i = <rb_i, permit> \land \{\neg rb_j \text{ for any permit rule } r_j = <rb_j, permit> \text{ for } j \, != i\})$

b) *deny-unless-permit*:

i) <u>*Reachability constraint*</u>

The policy target should be *true or* should evaluate to an error. For any *permit* rule $r_j = <rb_j, permit> (j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rule with *permit* effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j < i\}$

ii) <u>*Necessity constraint*</u>

The rule should evaluate to *true*. Formally, rb_i

iii) <u>*Propagation constraint*</u>

If rule under consideration is *deny* rule, then the mutant is equivalent because the rule level decision in mutant will be *N/A* and, in original policy, it will be *true* or *error. However,* for *deny-unless-permit* RCA, anything other than *permit* effect results in *deny* decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both will behave the same.

Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect evaluate to *N/A* or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RER when RCA is *deny-unless-permit*.

$(\{PT \vee Error(PT)\} \wedge \{rb_i\} \wedge \{\neg rb_j \vee Error(rb_i)$ for any permit rule $r_j = < rb_j,$ permit $>$ for $j \,!= i\})$

c) *first-applicable*:

i) <u>*Reachability constraint*</u>

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A*.

ii) <u>*Necessity constraint*</u>

If policy target evaluates to *true*, the necessity constraint is that – rule body evaluates to *N/A* or *error*.

If policy target evaluates to an error, the necessity constraint is that – rule body evaluates to *N/A*.

iii) <u>*Propagation constraint*</u>

If policy target evaluates to *true* and i^{th} rule body evaluates to *true*, then for all other rules which have the same effect as that of ith rule should evaluate to *N/A* or *error*.

If policy target evaluates to *true* and i^{th} rule body evaluates to an error, then for all other rules after the ith rule which has the same effect as that of ith rule should evaluate to *N/A* or *true*

If policy target evaluates to an error and i^{th} rule body evaluates to an error *or true*, then for all other rules after the ith rule which has the same effect as that of ith rule should evaluate to *N/A*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RER when RCA is *first-applicable*.

$(\{PT\} \wedge rb_i \wedge \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j$ such that $re_i = re_j \text{ for } j > i\})$

$\vee$

$(\{PT\} \wedge Error(rb_i) \wedge \{\neg rb_j \vee rb_j \text{ for any rule } r_j \text{ for } j < i\} \wedge \{\neg rb_j \text{ for any rule } r_j$ such that $re_i = re_j \text{ for } j > i\})$

$\vee$

$(\{Error(PT)\} \wedge (rb_i \vee Error(rb_i)) \wedge \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \wedge \{\neg rb_j \text{ for any rule } r_j$ such that $re_i = re_j \text{ for } j > i\})$

3.9 FDC for Policy Target True (PTT)

Policy target *true* is a mutation operator which alters the policy target such that it will always evaluate to *true*. One of the transformation rules to make target always evaluate to *true* is to make it empty so that it will always evaluate to *true*. Since it has a fault in the target of a policy, it is under the category incorrect policy target.

Consider an XACML policy *P* = <*PT, RCA, RL*> and *P'* = <*PT', RCA, RL'*> where *PT and PT'* are such that *P'* is similar to *P* except the target *PT'* of *P'* always evaluates to *true*. Here, *P'* is called the Policy Target True (PTT) Mutant of *P*. The fault detection condition for Policy Target True based on RCA are given below.

a) permit-overrides:

i) Reachability constraint

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) Necessity constraint

The policy target evaluates to *N/A* or *error*.

iii) <u>*Propagation constraint*</u>

If policy target evaluates to *N/A*, at least one rule should evaluate to *true* or *error*.

If policy target evaluates to an error, there should be one permit rule that evaluates to *true* or all permit rules evaluate to *N/A* and at least one deny rule evaluates to *true*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of PTT when RCA is *permit-overrides:*

$(\neg PT \wedge \{rb_j \vee Error(rb_j)\ \textit{for any rule}\ r_j\}) \vee (Error(PT) \wedge (\{rb_j\ \textit{for any rule}\ r_j\ \textit{such that}\ r_j = <rb_j, permit>\} \vee \{\neg rb_j\ \textit{for all rule}\ r_j\ \textit{such that}\ r_j = <rb_j, permit> \wedge rb_j\ \textit{for any rule}\ r_j\ \textit{such that}\ r_j = <rb_j, deny>))$

***b)* deny-unless-permit:**

i) <u>*Reachability constraint*</u>

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) <u>*Necessity constraint*</u>

The policy target evaluates to *N/A* or *error*.

iii) <u>*Propagation constraint*</u>

Necessity constraint is sufficient for fault detection. Combining all constraints as a single constraint, we get the following constraint for fault detection of PTT when RCA is *deny-unless-permit*: $\neg PT \vee Error(PT)$

***c)* first-applicable:**

i) <u>*Reachability constraint*</u>

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) *Necessity constraint*

The policy target evaluates to *N/A* or *error*.

iii) *Propagation constraint*

If policy target evaluates to *true*, at least one rule should evaluate to *true* or *error*.

If policy target evaluates to an error, there should be one rule that evaluates to *true* and all other rules before that rule should evaluate to *N/A*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of PTT when RCA is *first-applicable*:

$(\neg PT \wedge \{rb_j \vee Error(rb_j) \text{ for any rule } r_j\}) \vee (Error(PT) \wedge (\{rb_j \text{ for any rule } r_j \wedge \neg rb_i \text{ for any rule } r_i \text{ such that } i < j\}))$

3.10 FDC for Policy Target False (PTF)

Policy target *false* is a mutation operator which alters the policy target such that it will always evaluate to *false*. Since it has a fault in the target of a policy, it is under the category incorrect policy target.

Consider an XACML policy *P* = <*PT, RCA, RL*> and *P'* = <*PT', RCA, RL'*> where *PT and PT'* are such that *P'* is similar to *P* except the target *PT'* of *P'* always evaluates to *false*. Here, *P'* is called the Policy Target False (PTF) Mutant of *P*. The fault detection condition for Policy Target False based on RCA are given below.

a) permit-overrides:

i) *Reachability constraint*

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) *Necessity constraint*

The policy target evaluates to *true* or *error*.

iii) *Propagation constraint*

At least one rule should evaluate to *true* or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of PTF when RCA is *permit-overrides:*

$((\neg PT \vee Error(PT)) \wedge \{rb_j \vee Error(rb_j)\ for\ any\ rule\ r_j\}$

b) ***deny-unless-permit*:**

i) *Reachability constraint*

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) *Necessity constraint*

The policy target evaluates to *true* or *error*.

iii) *Propagation constraint*

Necessity constraint is sufficient for fault detection. Combining all constraints as a single constraint, we get the following constraint for fault detection of PTT when RCA is *permit-overrides*: PT ∨ Error(PT)

c) ***first-applicable*:**

i) *Reachability constraint*

The policy target is the first element to be evaluated in a policy element. Hence, it is empty.

ii) *Necessity constraint*

The policy target evaluates to *true* or *error*.

iii) *Propagation constraint*

At least one rule should evaluate to *true* or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of PTT when RCA is *permit-overrides:*

$((\neg PT \vee Error(PT)) \wedge \{rb_j \vee Error(rb_j)\ \text{for any rule}\ r_j\}$

3.11 FDC for First Permit Rule (FPR)

First Permit Rule is a mutation operator which changes the position of the first Deny rule with the first Permit rule that follows the first Deny rule. Since it has a fault with the ordering of the rules, it is under the category incorrect rule ordering.

Consider an XACML policy P = <PT, RCA, RL> where RL = $<r_1, \ldots, r_d \ldots r_p .. r_n>$, RL' = $<r_1, ..., r_p, \ldots r_d, ..., r_n>$ such that P' is similar to P except the order of first permit rule following the first deny rule in P is swapped in P'.

Note: The order of permit rules and deny rules does not alter the decision of the policy element if the RCA are Permit-Override, Deny Override, Permit-unless-Deny or Deny-unless-Permit

The order of permit rules and deny rules matters only when both or at least one pair of permit rule and deny rule can be applied (true or error) at the same time. If they cannot be applied at the same time, then such a mutant will be equivalent to the original policy.

The fault detection condition of FPR when RCA is *first-applicable* is as follows:

i) *Reachability constraint*

The policy target evaluates to *true* or *error.*

ii) *Necessity constraint*

The rule with first *permit* effect should evaluate to *true* or *error* and the rule with first *deny* effect following the first *permit* rule should evaluate to *true* or *error.*

iii) *Propagation constraint*

Necessity constraint is sufficient for fault detection.

$(rb_p \vee Error(rb_p)) \wedge (rb_d \vee Error(rb_d))$ for the first permit rule r_p = <rb_p, permit> and for the first deny rule r_d = <rb_d, deny> after r_p.

3.12 FDC for First Deny Rule (FDR)

First Deny Rule is a mutation operator which changes the position of the first Permit rule with the first Deny rule that follows the first Permit rule. Since it has a fault with the ordering of the rules, it is under the category incorrect rule ordering.

Consider an XACML policy P = <PT, RCA, RL> where RL = <r_1, …. ,r_d… r_p..r_n>, RL' = <r_1, ..., r_p, … r_d, ..., r_n> such that P' is similar to P except the order of first deny rule following the first permit rule in P is swapped in P'.

The fault detection condition of FDR when RCA is *first-applicable* is as follows:

i) *Reachability constraint*

The policy target evaluates to *true* or *error.*

ii) *Necessity constraint*

The rule with first *deny* effect should evaluate to *true* or *error* and the rule with first *permit* effect following the first *deny* rule should evaluate to *true* or *error.*

iii) *Propagation constraint*

Necessity constraint is sufficient for fault detection.

$(rb_d \vee Error(rb_d)) \wedge (rb_p \vee Error(rb_p))$ for the first deny rule r_d = <rb_d, deny> and for the first permit rule r_p = <rb_p, permit> after r_d.

3.13 FDC for Remove Parallel Target Element (RPTE)

Remove Parallel Target Element is a mutation operator in which an *AnyOf* element of the target or *AllOf* element of an *AnyOf* element is removed from the target. Since it has a fault of missing a target element, it is under the category missing target element.

Let PT be the target element of Policy P of the rule r_i. The target element is composed of the conjunction of *AnyOf* element and each *AnyOf* element is composed of the disjunction of *AllOf* elements. Hence PT = $AnyOf_1 \wedge AnyOf_2 \wedge \ldots AnyOf_i \wedge \ldots AnyOf_n$ and arbitrary i^{th} $AnyOf_i = AllOf_{i1} \vee AllOf_{i2} \vee AllOf_{ij} \vee AllOf_{im}$. If either i^{th} *AnyOf* is removed or j^{th} *AllOf* from i^{th} *AnyOf* is removed, the resulted faulty policy is called Remove parallel target element mutant. The fault detection condition for RPTE is presented as follows: -

a) Permit-overrides:

i) Reachability constraint

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j < i\}$

ii) Necessity constraint

The necessity constraint for RPTE with i^{th} *AnyOf* element missing is that – rule target expression should have negation term for i^{th} *AnyOf* element and rule condition evaluates to *true*.

Formally, $(AnyOf_1 \wedge \neg \mathbf{AnyOf_i} \wedge .. AnyOf_n) \wedge rc_i$

The necessity constraint for RPTE with j^{th} *AllOf* element of ith *AnyOf* element missing is that – rule target expression should have a positive term for j^{th} *AllOf* element of ith *AnyOf*

element and all other *AllOf* element should have a negative term and rule condition evaluates to *true*.

Formally, $(AnyOf_1 \wedge \boldsymbol{AnyOf_i = (\neg AllOf_1 \vee AllOf_j \vee \neg AllOf_m)} \wedge .. AnyOf_n) \wedge rc_i$

iii) <u>*Propagation constraint*</u>

If policy target evaluates to *true* and i^{th} rule effect is *deny*, then all other rules should evaluate to *N/A* or *error*. Further, there should not exist a pair of rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$(\{PT)\} \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j \text{ for } j \mathrel{!}= i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

If policy target evaluates to an error, and i^{th} rule effect is *deny*, then all other rules with *deny* effect should evaluate to *N/A* and all other rules with *permit* effect should not evaluate to *true*. Further, there should not exist a pair of rules (excluding the current rule under consideration) such that one of them is *permit* rule which evaluates to an error and other is *deny* rule which evaluates to *true* or *error*.

$(\{Error(PT)\} \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \mathrel{!}= i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j \mathrel{!}= i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\})$

If policy target evaluates to *true*, and i^{th} rule effect is *permit*, then all other permit rules should evaluate to *N/A* or *error*.

$(\{PT\} \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \mathrel{!}= i\})$

If policy target evaluates to an error, and i^{th} rule effect is *permit*, then all other rules with *permit* effect should evaluate to *N/A* if all *deny* rules evaluate to *N/A*. However, if any *deny* rule evaluates to *true* or *error*, then another permit effect can evaluate to *N/A* or *error*.

$$(\{Error(PT)\} \wedge r_i = <rb_i, permit> \wedge [\{\neg rb_j \text{ for any rule } r_j \text{ for } j \mathrel{!=} i\} \vee \{ \exists d\ (rb_d \vee Error(rb_d)) \wedge (r_d = <rb_d, deny>) \wedge (\neg rb_j \vee Error(rb_j)) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \mathrel{!=} i \}])$$

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is *permit-overrides:*

$$(\{PT \} \wedge \{(AnyOf_1 \wedge \neg \mathbf{AnyOf_i} \wedge ..\ AnyOf_n) \vee (AnyOf_1 \wedge \mathbf{AnyOf_i = (\neg AllOf_1 \vee AllOf_j \vee \neg AllOf_m)} \wedge ..\ AnyOf_n)\} \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j \text{ for } j \mathrel{!=} i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\}\})$$

$\vee$

$$(\{Error(PT)\} \wedge \{(AnyOf_1 \wedge \neg \mathbf{AnyOf_i} \wedge ..\ AnyOf_n) \vee (AnyOf_1 \wedge \mathbf{AnyOf_i = (\neg AllOf_1 \vee AllOf_j \vee \neg AllOf_m)} \wedge ..\ AnyOf_n)\} \wedge rc_i \wedge r_i = <rb_i, deny> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \mathrel{!=} i\} \wedge \{\neg rb_j \text{ for any deny rule } r_j = <rb_i, deny> \text{ for } j \mathrel{!=} i\} \wedge \{\neg(\exists(p,d) \text{ such that } (i \neq p \neq d) \wedge (r_d = < rb_d, deny>) \wedge (r_p = < rb_p, permit>) \wedge Error(rb_p) \wedge (Error(rb_d) \vee rb_d))\}\})$$

$\vee$

$$(\{PT \} \wedge \{(AnyOf_1 \wedge \neg \mathbf{AnyOf_i} \wedge ..\ AnyOf_n) \vee (AnyOf_1 \wedge \mathbf{AnyOf_i = (\neg AllOf_1 \vee AllOf_j \vee \neg AllOf_m)} \wedge ..\ AnyOf_n)\} \wedge rc_i \wedge r_i = <rb_i, permit> \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any permit rule } r_j = <rb_i, permit> \text{ for } j \mathrel{!=} i\})$$

$\vee$

$(\{Error(PT)\} \wedge \{(AnyOf_1 \wedge \neg \mathbf{AnyOf_i} \wedge .. AnyOf_n) \vee (AnyOf_1 \wedge \mathbf{AnyOf_i = (\neg AllOf_1 \vee AllOf_j \vee \neg AllOf_m)} \wedge .. AnyOf_n)\} \wedge rc_i \wedge r_i = <rb_i, permit> \wedge [\{\neg rb_j$ *for any rule* r_j *for* j $!= i\} \vee \{ \exists d (rb_d \vee Error(rb_d)) \wedge (r_d = <rb_d, deny>) \wedge (\neg rb_j \vee Error(rb_j))$ *for any permit rule* $r_j = <rb_i, permit>$ *for* $j != i \}])$

b) *deny-unless-permit*:

i) <u>Reachability constraint</u>

The policy target should be *true or* should evaluate to an error. For any *permit* rule $r_j = <rb_j, permit> (j<i)$ before rule r_i, rb_j should not evaluate to *true* (i.e it should be *N/A* or evaluates to an error).

Hence, the reachability constraint is - policy target evaluates to *true* or *error* and all the previous rules with *permit* effect before the current rule under consideration should evaluate to *N/A* or *error* (i.e should not evaluate to *true*).

In formal notation, it will be $\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)$ *for any permit rule* $r_j = <rb_j, permit>$ *for* $j < i\}$

ii) <u>Necessity constraint</u>

The necessity constraint for RPTE with i^{th} *AnyOf* element missing is that – rule target expression should have negation term for i^{th} *AnyOf* element and rule condition evaluates to *true*.

Formally, $(AnyOf_1 \wedge \neg \mathbf{AnyOf_i} \wedge .. AnyOf_n) \wedge rc_i$

The necessity constraint for RPTE with j^{th} *AllOf* element of ith *AnyOf* element missing is that – rule target expression should have a positive term for j^{th} *AllOf* element of ith *AnyOf* element and all other *AllOf* element should have a negative term and rule condition evaluates to *true*.

Formally, $(AnyOf_1 \wedge \mathbf{AnyOf_i = (\neg AllOf_1 \vee AllOf_j \vee \neg AllOf_m)} \wedge .. AnyOf_n) \wedge rc_i$

iii) *Propagation constraint*

If rule under consideration is *deny* rule, then the mutant is equivalent because the rule level decision in mutant will be deny, and, in original policy, it will be N/A or error. However, for *deny-unless-permit* RCA, anything other than *permit* effect results in *deny* decision. And, it is only the current rule under consideration where mutant differs from original policy and hence both of them will behave the same.

Hence, the rule under consideration should be *permit* rule such that all the rules with *permit* effect after the ith rule should evaluate to *N/A* or *error*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is *deny-unless-permit*.

$(\{PT \vee Error(PT)\} \wedge \{(\{(AnyOf_1 \wedge \mathbf{\neg AnyOf_i} \wedge .. AnyOf_n) \vee (AnyOf_1 \wedge \mathbf{AnyOf_i = (\neg AllOf_1 \vee AllOf_j \vee \neg AllOf_m)} \wedge .. AnyOf_n)\}) \wedge rc_i\} \wedge \{\neg rb_j \vee Error(rb_i) \text{ for any permit rule } r_j = < rb_j, permit > \text{for } j \,!= i\})$

c) ***first-applicable*:**

i) *Reachability constraint*

Policy target should evaluate to *true* or *error* and all the previous rules with permit effect before the current rule under consideration should evaluate to *N/A*.

ii) *Necessity constraint*

The necessity constraint for RPTE with i^{th} *AnyOf* element missing is that – rule target expression should have negation term for i^{th} *AnyOf* element and rule condition evaluates to *true*.

Formally, $(AnyOf_1 \wedge \mathbf{\neg AnyOf_i} \wedge .. AnyOf_n) \wedge rc_i$

The necessity constraint for RPTE with j^{th} *AllOf* element of ith *AnyOf* element missing is that – rule target expression should have a positive term for j^{th} *AllOf* element of ith *AnyOf* element and all other *AllOf* element should have a negative term and rule condition evaluates to *true*.

Formally, $(AnyOf_1 \wedge \mathbf{AnyOf_i = (\neg AllOf_1 \vee AllOf_j \vee \neg AllOf_m)} \wedge .. AnyOf_n) \wedge rc_i$

iii) *Propagation constraint*

If policy target evaluates to *true*, then for all other rules which have the same effect as that of ith rule should evaluate to *N/A* or *error*.

If policy target evaluates to an error, then for all other rules which have the same effect as that of ith rule should evaluate to *N/A*.

Combining all constraints as a single constraint, we get the following constraint for fault detection of RTT when RCA is *first-applicable*.

$(\{PT\} \wedge (\{(AnyOf_1 \wedge \mathbf{\neg AnyOf_i} \wedge .. AnyOf_n) \vee (AnyOf_1 \wedge \mathbf{AnyOf_i = (\neg AllOf_1 \vee AllOf_j \vee \neg AllOf_m)} \wedge .. AnyOf_n)\}) \wedge rc_i \wedge \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \wedge \{\neg rb_j \vee Error(rb_j) \text{ for any rule } r_j$ such that $re_i = re_j \text{ for } j > i\})$

$\vee$

$(Error(PT)\} \wedge (\{(AnyOf_1 \wedge \mathbf{\neg AnyOf_i} \wedge .. AnyOf_n) \vee (AnyOf_1 \wedge \mathbf{AnyOf_i = (\neg AllOf_1 \vee AllOf_j \vee \neg AllOf_m)} \wedge .. AnyOf_n)\}) \wedge rc_i \wedge \{\neg rb_j \text{ for any rule } r_j \text{ for } j < i\} \wedge \{\neg rb_j \text{ for any rule } r_j$ such that $re_i = re_j \text{ for } j > i\})$

3.14 FDC for Change Rule Combining Algorithm (CRC)

Change Rule Combining Algorithm is a mutation operator in which the combining algorithm of the policy is changed to another combining algorithm. Since it has a fault of in combining algorithm, it is under the category incorrect combining algorithm.

Consider an XACML policy P = <PT, RCA, RL> and P' = <PT, RCA', RL> such that P' is similar to P except the RCA' in P' is different from one in P. Here, P' is called the Change Combining Algorithm (CCA) Mutant of P. The fault detection condition for CRC presented as follows.

a) FDC for change from *permit-overrides* to *Deny-overrides*:

Policy target should evaluate to *true* or *error.*

There should exist at least one pair of rules such that one of them is *permit* rule and another is *deny* such that both evaluate to *true* or one evaluates to *true* and other evaluates to an error.

Formally,

$(PT \vee Error(PT)) \wedge \exists(p,d)$ *such that* $r_p = <rb_p, permit> \wedge r_d = <rb_d, deny> \wedge (\{rb_p \wedge rb_d\} \vee \{Error(rb_p) \wedge rb_d\} \vee \{Error(rb_d) \wedge rb_p\})$

Note: if all of the permit rules are N/A while one or more deny rules evaluate to true or error and if all of the deny rules are N/A while one or more permit rules evaluate to true or error, permit-overrides and deny-overrides behaves the same.

b) FDC for change from *permit-overrides* to *deny-unless-permit*:

Policy target should evaluate to *true* or *error.*

$\wedge$

(All of the rules evaluate to *N/A*

$\vee$

one or more *permit* rules evaluates to an error

$\vee$

one or more *deny* rules evaluates to an error and the rest are *N/A* and *PT* evaluates to *true*)

Formally,

(PT $\vee$ *Error(PT))* $\wedge$ *({*$\neg rb_j$ *for all rules in P}* $\vee$ *{Error(*rb_j*) for any permit rule* r_j = <rb_j*, permit*> *in P}* $\vee${PT $\wedge$ *Error(*rb_k*) for some deny rule* r_k = <rb_k*, deny*> *in P* $\wedge$ $\neg$ rb_j *for other rules* r_j *in P such that* $k \neq j$})

c) FDC for change from *permit-overrides* to *permit-unless-deny*:

Policy target should evaluate to *true* or *error.*

$\wedge$

(All of the rules evaluate to *N/A*

$\vee$

one or more *permit* rule evaluates to *true* only if one or more *deny* rules evaluate to *true*.

$\vee$

one or more *deny* rule evaluates to *true* only if one or more *permit* rule evaluates to *true* or *error*

$\vee$

one or more *permit* rule evaluates to an error and the rest evaluates to *N/A* and PT evaluates to *true*.)

Formally,

(PT $\vee$ *Error(PT))* $\wedge$ *({*$\neg rb_j$ *for all rules in P}* $\vee$ *{* rb_d $\wedge$ *(*rb_p $\vee$ Error(rb_p *)) for any permit rule* r_p = <rb_p*, permit*> *and for any deny rule* r_d = <rb_d*, deny*> *in P}* $\vee${PT $\wedge$ *Error(*rb_k*) for some permit rule* r_k = <rb_k*, permit*> *in P* $\wedge$ $\neg$ rb_j *for other rules* r_j *in P such that* $k \neq j$})

d) FDC for change from *permit-overrides* to *first-applicable*:

Policy target should evaluate to *true* or *error.*

$\wedge$

if r_f is the first rule that evaluates to *true,* then r_f is *deny* rule and there exists one or more *permit* rule that evaluates to *true* or *error*

$\vee$

if r_f is the first rule that evaluates to an error and r_f is *deny* rule, then there exists one or more *permit* or *deny* rule that evaluates to *true* or one or more *permit* rule that evaluates to an error

$\vee$

if r_f is the first rule that evaluates to an error and r_f is *permit* rule, then there exists one or more *permit* or *deny* rule that evaluates to *true* or one or more *deny* rule that evaluates to an error)

Formally,

$(PT \vee Error(PT))$

$\wedge$

$(\{(\neg rb_j \wedge rb_f \text{ for some } f \text{ such that } j < f) \wedge (r_f = <rb_f, deny> \wedge rb_p \text{ for some permit rule } r_p = <rb_p, permit>)\}$

$\vee$

$\{(\neg rb_j \wedge Error(rb_f) \text{ for some } f \text{ such that } j < f) \wedge (r_f = <rb_f, deny>) \wedge (rb_i \text{ for some permit or deny rule} \vee \text{Error}(rb_p) \text{ for some } permit \text{ rule } rb_p)))\}$

$\vee$

$\{(\neg rb_j \wedge Error(rb_f) \text{ for some } f \text{ such that } j < f) \wedge (r_f = <rb_f, permit>) \wedge (rb_i \text{ for some permit or deny rule} \vee Error(rb_d) \text{ for some } deny \text{ rule } rb_d))\})$

e) FDC for change from *deny-overrides* to *permit-unless-deny*:

Policy target should evaluate to *true* or *error.*

$\wedge$

(All of the rules evaluate to *N/A*

$\vee$

one or more *deny* rule evaluates to an error

$\vee$

one or more *permit* rule evaluates to an error and rest are *N/A* and *PT* evaluates to *true*)

Formally,

$(PT \vee Error(PT)) \wedge (\{\neg rb_j \text{ for all rules in } P\} \vee \{Error(rb_j) \text{ for any deny rule } r_j = \langle rb_j, deny\rangle \text{ in } P\} \vee \{PT \wedge Error(rb_k) \text{ for some permit rule } r_k = \langle rb_k, permit\rangle \text{ in } P \wedge \neg rb_j \text{ for other rules } r_j \text{ in } P \text{ such that } k \neq j\})$

f) FDC for change from *deny-overrides* to *deny-unless-permit*:

Policy target should evaluate to *true* or *error.*

$\wedge$

(All of the rules evaluate to *N/A*

$\vee$

one or more *deny* rule evaluates to *true* only if one or more *permit* rules evaluate to *true*.

$\vee$

one or more *permit* rule evaluates to *true* only if one or more *deny* rule evaluates to *true* or *error*

$\vee$

one or more *deny* rule evaluates to an error and rest evaluates to *N/A* and PT evaluates to *true*.)

Formally,

$(PT \vee Error(PT)) \wedge (\{\neg rb_j$ *for all rules in P*$\} \vee \{ rb_p \wedge (rb_d \vee$ Error$(rb_d))$ *for any permit rule* r_p = <rb_p, *permit*> *and for any deny rule* r_d = <rb_d, *deny*> *in P*$\} \vee \{$PT $\wedge$ $Error(rb_k)$ *for some deny rule* r_k = <rb_k, *deny*> *in P* $\wedge \neg rb_j$ *for other rules* r_j *in P such that* $k \neq j\})$

g) FDC for change from *deny-overrides* to *first-applicable*:

Policy target should evaluate to *true* or *error.*

$\wedge$

if r_f is the first rule that evaluates to *true,* then r_f is *permit* rule and there exists one or more *deny* rule that evaluates to *true* or *error*

$\vee$

if r_f is the first rule that evaluates to an error and r_f is *permit* rule, then there exists one or more *permit* or *deny* rule that evaluates to *true* or one or more *deny* rule that evaluates to an error

$\vee$

if r_f is the first rule that evaluates to an error and r_f is *deny* rule, then there exists one or more *permit* or *deny* rule that evaluates to *true* or one or more *permit* rule that evaluates to an error

Formally,

$(PT \vee Error(PT))$

$\wedge$

$(\{(\neg rb_j \wedge rb_f$ *for some f such that* $j < f) \wedge (r_f$ = <rb_f, *permit*> $\wedge\ rb_d$ *for some deny rule* r_d = <rb_d, *deny*>$)\}$

$\vee$

$\{(\neg rb_j \wedge Error(rb_f)$ *for some f such that* $j < f) \wedge (r_f = \langle rb_f, permit\rangle) \wedge (rb_i$ *for some permit or deny rule* $\vee$ Error(rb_d) for some *deny* rule $rb_d)))\}$

$\vee$

$\{(\neg rb_j \wedge Error(rb_f)$ *for some f such that* $j < f) \wedge (r_f = \langle rb_f, deny\rangle) \wedge (rb_i$ *for some deny or permit rule* $\vee$ *Error*(rb_p) for some *permit* rule $rb_p))\})$

h) FDC for change from *permit-unless-deny* to *deny-unless-permit*:

Policy target should evaluate to *true* or *error.*

$\wedge$

(All of the rules evaluate to *N/A* or *error*

$\vee$

if one rule evaluates to *true,* then other rules with opposite effect must evaluate to *true*.)

Formally,

$(PT \vee Error(PT)) \wedge (\{\neg rb_j \vee Error(rb_j)$ *for all rules in P*$\} \vee \{ rb_p \wedge rb_d$ *for any permit rule* $r_p = \langle rb_p, permit\rangle$ *and for any deny rule* $r_d = \langle rb_d, deny\rangle$ *in P*$\})$

i) FDC for change from *permit-unless-deny* to *first-applicable*:

Policy target should evaluate to *true* or *error*

$\wedge$

if r_f is the first rule that evaluates to *true,* then r_f is *permit* rule and there exists one or more *deny* rule that evaluates to *true*.

$\vee$

if r_f is the first rule that evaluates to an error and r_f is *permit* rule*,* then policy target should evaluate to *true* or one or more *deny* rule should evaluate to *true*.

$\vee$

if r_f is the first rule that evaluates to an error and r_f is *deny* rule, then policy target should evaluate to *true* if one or more *deny* rule evaluates to *true*.)

Formally,

$(PT \vee Error(PT))$

$\wedge$

$(\{(\neg rb_j \wedge rb_f \text{ for some } f \text{ such that } j < f) \wedge (r_f = \langle rb_f, permit \rangle \wedge rb_d \text{ for some deny rule } r_d = \langle rb_d, deny \rangle)\}$

$\vee$

$\{(\neg rb_j \wedge Error(rb_f) \text{ for some } f \text{ such that } j < f) \wedge (r_f = \langle rb_f, permit \rangle \wedge (rb_d \text{ for some deny rule } r_d = \langle rb_d, deny \rangle) \vee PT)\}$

$\vee$

$\{(\neg rb_j \wedge Error(rb_f) \text{ for some } f \text{ such that } j < f) \wedge (r_f = \langle rb_f, deny \rangle \wedge (\neg (rb_d \text{ for some deny rule } r_d = \langle rb_d, deny \rangle)) \vee PT)\})$

j) FDC for change from *deny-unless-permit* to *first-applicable*:

Policy target should evaluate to *true* or *error*

$\wedge$

if r_f is the first rule that evaluates to *true*, then r_f is *deny* rule and there exists one or more *permit* rule that evaluates to *true*.

$\vee$

if r_f is the first rule that evaluates to an error and r_f is *deny* rule, then policy target should evaluate to *true* or one or more *permit* rule should evaluate to *true*.

$\vee$

if r_f is the first rule that evaluates to an error and r_f is *permit* rule, then policy target should evaluate to *true* if one or more *permit* rule evaluates to *true*.)

Formally,

$(PT \vee Error(PT))$

$\wedge$

$(\{(\neg rb_j \wedge rb_f$ *for some f such that* $j < f) \wedge (r_f = \langle rb_f, deny \rangle \wedge rb_p$ *for some permit rule* $r_p = \langle rb_p, permit \rangle)\}$

$\vee$

$\{(\neg rb_j \wedge Error(rb_f)$ *for some f such that* $j < f) \wedge (r_f = \langle rb_f, deny \rangle \wedge (rb_p$ *for some permit rule* $r_p = \langle rb_p, permit \rangle) \vee PT)\}$

$\vee$

$\{(\neg rb_j \wedge Error(rb_f)$ *for some f such that* $j < f) \wedge (r_f = \langle rb_f, permit \rangle \wedge (\neg (rb_p$ *for some permit rule* $r_p = \langle rb_p, permit \rangle)) \vee PT)\})$

Note: FDC for CRC from permit-overrides to deny-overrides is applicable from deny-overrides to permit-overrides as well. Hence, we have all possible combinations between five RCAs.

CHAPTER 4

Mutation-Based Test Generation

Mutation-based test generation involves generating test suites exploiting the fault detection condition with the goal to kill all the non-equivalent mutants. The formalization of complete fault detection conditions makes it feasible to automatically perform strong mutation-based test generation. From fault detection condition, we know that if any request satisfies reachability, necessity and propagation for a fault, it will produce a different result in original policy, and faulty policy. This difference in response can be exploited to distinguish faulty policy from the correct policy. Hence, the goal is to generate test input that satisfies the three constraints of the fault detection condition - reachability, necessity and sufficiency/propagation for each possible fault policy.

FDC may include multiple mutually exclusive conditions. If one of them is satisfied, it will be sufficient for the fault detection. As a result, we just use one of the mutually exclusive conditions for each fault. In other words, we do not need to generate multiple requests for each of the mutually exclusive conditions because it will just generate redundant test cases. The process of identifying such sufficient mutually exclusive conditions (if any exist) to detect a fault is discussed in Section 1.3.2. The overall process for strong mutation-based (SMT) test generation is depicted in Algorithm 1.

4.1 Strong Mutation-Based Test Generation

Algorithm 1: Mutation-based generation of the near-optimal test suite (SMT)

Import functions: kill(M, Q) returns the list of mutants in *M* that are killed by test suite *Q*

Z3-request(FDC) returns a solution to the constraint *FDC*

Input: Policy $P = < PT, RCA, [r_1, r_2, \ldots, r_n] >$, *Ω* is a list of mutation operators

Output: A set of access requests *Q*

Variables: *M* is a mutant pool, *OPS* is a list of mutation operators, *FDC* is a fault detection condition, *q* is a test input

1 $Q \leftarrow \epsilon$

2 *While* $\Omega \neq \varnothing$

3 *OPS* ← *select one or more mutation operators from Ω*

4 *M* ← *list of mutants created by mutation operators OPS*

5 $M \leftarrow M - kill(M, Q)$

6 *While* $M \neq \varnothing$

7 *FDC* ← *compose the fault detection conditions of one or more mutants*

8 $q \leftarrow$ *Z3-request(FDC)*

9 *If (q is not null) // otherwise equivalent mutant*

10 $Q \leftarrow Q \cup \{q\}$

11 $M \leftarrow M - kill(M, \{q\})$

12 *EndIf*

13 *EndWhile*

14 *EndWhile*

15 *Return Q*

From formalization of the fault detection condition, we have identified RTF, RCF, ANF, RNF, and RER have common fault detection conditions among various possible mutually exclusive fault detection conditions for each of them. Algorithm 6 in Section 4.3 presents such common fault detection conditions among them. As a result, we do not need to deal with each of those mutation operators individually. In other words, if we generate a request that satisfies that common fault detection condition among them, it can kill the mutants from RTF, RCF, ANF, RNF and RER. So, we select one (or more mutation operators if they have common FDC) at a time and generate mutants for them (line 3-4), and run the mutants against the existing tests. The mutants that are already killed by the existing tests are removed from the mutant pool (line 5). Then we compose the constraint for one or more compatible mutants (line 7) and then solve the constraint by using the Z3 constraint solver [17] (line 8) If the constraint is solved, we convert the solution into an access request and add it to the test suite (lines 9-10), otherwise, the mutant is considered an equivalent mutant. We also run the new test against the current mutant pool. Mutants killed by the new test will be removed from the pool (line 11). When all mutation operators are handled, we return Q as the generated test suite.

Algorithm 1 is computationally expensive for the policy with a large number of the rules because of the involved optimization (step 4, 5 and 11) which needs to generate mutants and determine whether they are killed or not. These optimization steps are costly because *kill(M,Q)* is in the order of $O(n^3)$ where n is the number of the rule in a policy. Executing a test with each mutant has a complexity of $O(n)$. The number of mutants is in linear order and test suite size is linear as well. Hence, *kill(M,Q)* is in the order of $O(n^3)$

because in the worst case we need to run an operation with the complexity of $O(n)$ for mutants of the size $O(n)$ for each request in test suite of the size $O(n)$.

As a result, if n grows very large in big policies, the optimization steps grows in the cubic order which makes it unfeasible to use for large policies. So, we also formulated the mutation-based test generation without optimization which we referred to as Non-Optimized Strong Mutation-based Testing (*NO-SMT*) which generates a test suite that achieves a 100% mutant score like SMT. NO-SMT is nothing but the Algorithm 1 itself excluding the steps 4,5 and 11. However, it will have many redundant test cases than that from SMT. Hence, it is the trade-off between test generation time against test suite size. We have implemented these algorithms as an extension to the open source XPA tool.

The complexity of step 7 (for a single iteration) i.e generation of FDC constraint will be in the order of $O(n)$. The details on FDC constraint generation for each mutation operator is presented in Section 4.1 through Section 4.14. The time complexity of step 8 (for single iteration) is the time complexity of Z3 for solving FDC constraint expression. Let, $O(Z3)$ represents the time complexity of Z3 at step 8. $kill(M, \{q\})$ in step 11 will be in the order of $O(n^2)$ since it is similar to $kill(M,Q)$ but for a single test q. Since the size of M will be in linear order, the time complexity of step 6 through step 13 will be in the order of $O(n.(O(Z3)+ n^2))$. Similarly, the time complexity of NO-SMT will be in the order of $O(n.(O(Z3) + n))$.

As demonstrated in the empirical study, the test generation time of *SMT* grows much faster than *RC, DC*, and *MC/DC.* It is worth pointing out that, in this paper, the mutation-based generation of the near-optimal test suite is not meant to be a practical test generation method for large-scale XACML policies. Instead, our goal is to use the test suite as a

benchmark for measuring relative cost-effectiveness of other testing methods. In addition, the mutation-based test suite can be used to help improve other testing methods by determining equivalent mutants and investigating fault detection conditions of live mutants. However, coverage-based test generation does not guarantee 100% mutation score and if 100% mutation score is desired irrespective of the size of a test suite, then we can use NO-SMT for the larger policies as well.

The following are algorithms used to construct fault detection condition for each possible mutant for each mutation operator in the fault model.

4.1.1 FDC Constraint and Tests Generation for CRE

Algorithm 2: *generateTestsForCRE(P)*

Input: Policy P= < PT, RCA, [r_1, r_2,…, r_n]>

Output: A set of access requests Q

1 $Q \leftarrow \epsilon$

2 *constraint* $\leftarrow$ *PT*

3 *For each Rule r_i in [r_1, r_2,…, r_n] in P, do*

4 *ruleConstraint* $\leftarrow$ *constraint*

5 *ruleConstraint* $\leftarrow$ *ruleConstraint* $\wedge$ *ruleReachability(P, r_i)*

6 *ruleConstraint* $\leftarrow$ *ruleConstraint* $\wedge$ $rt_i \wedge rc_i$

7 *ruleConstraint* $\leftarrow$ *ruleConstraint* $\wedge$ *rulePropagation(P,r_i,CRE)*

8 *Q* $\leftarrow$ *Q U Z3-request(ruleConstraint)*

9 *EndFor*

10 *return Q*

The input to the algorithm is policy $P = < PT, RCA, [r_1, r_2, \ldots, r_n]>$ and output is set of generated access request Q. Initially we set Q to be empty (line 1). We then set the constraint expression from policy target if it is present, otherwise, the constraint is empty (line 4). The next step is to iterate over each rule in the policy (line 3) and construct fault detection condition constraint to kill the mutant. For each rule, the rule constraint is set to constraint expression (line 4) from policy target constraint in line 2. Then, for each rule in the policy, the rule constraint expression is concatenated with reachability constraint (line 5). Once the reachability constraint is met, the next task is to concatenate the rule constraint with necessity constraint for the fault type (line 6) and then concatenate the propagation constraint (line 7). The constructed rule constraint is the constraint with sufficient constraint to kill the CRE mutant and is supplied to the constraint solver (line 8) to obtain the value for the test inputs. When all the rules are processed, the generated set of test input - Q is returned. The algorithm for rule reachability is presented in Algorithm 3 in Section 4.1.1 and that for rule propagation is presented in Algorithm 4 in Section 4.1.2. It should be noted that reachability and propagation constraint is constructed taking common mutually exclusive conditions for reachability and propagation constraint for RTT, RTF, RCT, RCF, ANF, RNF and RER such that it works for all of them. CRE has slightly different propagation constraint than others but still shares the common constraints among them. Hence, rather than defining redundant propagation constraint for CRE only, we used the same method for propagation constraint passing the mutation method itself as an argument to tweak the propagation constraint based on the mutation operator.

4.1.2 FDC Constraint for Rule Reachability

Algorithm 3: *ruleReachability(P, r_i)*

Import functions: hasCommonAttribute(rb_i, rb_j) → returns true if rule body of ith rule and rule body of jth rule has a common attribute

Input: Policy P= < PT, RCA, [r_1, r_2,…, r_n]> , Rule r_i= <rt_i, rc_i, re_i>

Output: constraint

```
1     constraint ← ϵ
2     If RCA = First-Applicable, then
3           For each rule r_k in [r_1, r_2,…, r_i-1] in P, do
4                 constraint ← constraint ∧ ¬ (rt_k ∧ rc_k)
5           endFor
6     else
7           if RCA = Permit-unless-Deny, then
8                 for each rule r_k in [r_1, r_2,…, r_i-1] in P, do
9                       if re_k = Deny, then
10                            constraint ← constraint ∧ ¬ (rt_k ∧ rc_k)
11                      endIF
12                endFor
13          else if RCA = Deny-unless-Permit, then
14                for each rule r_k in [r_1, r_2,…, r_i-1] in P, do
15                      if re_k = Permit, then
16                            constraint ← constraint ∧ ¬ (rt_k ∧ rc_k)
17                      endIf
```

```
18                endFor
19          else if RCA = Deny-overrides, then
20                for each rule r_k in [r_1, r_2,…, r_{i-1}] in P, do
21                      if re_k = Deny, then
22                            if hasCommonAttributes(rb_i,rb_k), then
24                                  constraint ← constraint ∧ ¬ (rt_k ∧ rc_k)
25                            else
26                                  dominantRuleCollection.add(r_k)
27                            endIf
28                      endIf
29                endFor
30          else if RCA = Permit-overrides, then
31                for each rule r_k in [r_1, r_2,…, r_{i-1}] in P, do
32                      if re_k = Permit, then
33                            if hasCommonAttributes(rb_i,rb_k), then
34                                  constraint ← constraint ∧ ¬ (rt_k ∧ rc_k)
35                            else
36                                  dominantRuleCollection.add(r_k)
37                            endIf
38                      endIf
39                endFor
40          endIf
41    endIf
```

4.2 Return Constraint

If rule combining algorithm is *first-applicable*, then all the previous rules before the current rule under consideration i.e. i^{th} rule should evaluate to *N/A* (line 2-5) otherwise if RCA is *permit-unless-deny*, then all the previous rules with deny effect should be *N/A* (line 7-12). If RCA is *deny-unless-permit*, all the previous rules with permit effect should be *N/A* (line 13-18). If RCA is *deny-overrides* (or *permit-overrides*), then we only falsify deny (or permit) rules with common attributes in rule body of current rule under consideration and mark the deny (or permit) rules with no common attribute for later to be used for propagation constraint (line 19-29 and line 30-41).

4.3 FDC Constraint for Propagation

Algorithm 4: *rulePropagation(P, r_i, mutationMethod)*

Input: Policy P= < PT, RCA, [r_1, r_2, ..., r_n]> , Rule r_i= <rt_i, rc_i, re_i>

Output: constraint

1 *constraint* $\leftarrow \epsilon$

2 *If RCA = First-Applicable, then*

3 *for each rule r_k in [r_{i+1}, r_{i+2}, ..., r_n] in P, do*

4 *if $re_k = re_i$, then*

5 *constraint* $\leftarrow$ *constraint* $\wedge \neg (rt_k \wedge rc_k)$

6 *endIf*

7 *endFor*

8 *else*

9 *if RCA = Permit-unless-Deny, then*

10 *for each rule r_k in [r_{i+1}, r_{i+2}, ..., r_n] in P, do*

```
11                  if re_k = Deny, then
12                      constraint ← constraint ∧ ¬ (rt_k ∧ rc_k)
13                  endIf
14              endFor
15          else if RCA = Deny-unless-Permit, then
16              for each rule r_k in [r_i+1, r_i+2,…, r_n] in P, do
17                  if re_k = Permit, then
18                      constraint ← constraint ∧ ¬ (rt_k ∧ rc_k)
19                  endIf
20              endIf
21          else if RCA = Deny-overrides, then
22              for each rule r_k in [r_i+1, r_i+2,…, r_n] in P, do
23                  if re_k = Deny, then
24                      if hasCommonAttributes(rb_i,rb_k), then
25                          constraint ← constraint ∧ ¬ (rt_k ∧ rc_k)
26                      else
27                          dominantRuleCollection.add(r_k)
28                      endIf
29                  endIf
30              endFor
31              if re_i = permit and mutationMethod ≠ CRE then
32                  for each rule r_l in dominantRuleCollection, do
33                      if ¬ (rt_l ∧ rc_l) → (rt_p ∧ rc_p) for r_p = <rb_p, permit>, then
```

```
34                          dominantIndeterminateFlag ← true
35                   else
36                        constraint ← constraint ∧ ¬ (rt_l ∧ rc_l)
37                   endIf
38              endFor
39
40              if dominantIndeterminateFlag = true, then
41                   permit-rules ← get permit rules of P
42                   for each r_p in permit-rules, do
43                        constraint ← constraint ∧ (rt_p ∧ rc_p)
44                   endFor
45              endIf
46         endIf
47    else if RCA = permit-overrides , then
48         for each rule r_k in [r_{i+1}, r_{i+2},…, r_n] in P, do
49              if re_k = permit, then
50                   if hasCommonAttributes(rb_i,rb_k), then
51                        constraint ← constraint ∧ ¬ (rt_k ∧ rc_k)
52                   else
53                        dominantRuleCollection.add(r_k)
54                   endIf
55              endIf
56         endFor
```

57 *if* re_i *= deny and mutationMethod* $\neq$ *CRE then*

58 *for each rule* r_l *in dominantRuleCollection, do*

59 *if* $\neg(rt_l \wedge rc_l) \rightarrow (rt_p \wedge rc_p)$ *for* $r_p = <rb_p, permit>$*, then*

60 *dominantIndeterminateFlag* $\leftarrow$ *true*

61 *else*

62 *constraint* $\leftarrow$ *constraint* $\wedge \neg(rt_l \wedge rc_l)$

63 *endIf*

64 *endFor*

65 *if dominantIndeterminateFlag = true, then*

66 *deny-rules* $\leftarrow$ *get deny rules of P*

67 *for each* r_d *in deny-rules, do*

68 *constraint* $\leftarrow$ *constraint* $\wedge (rt_d \wedge rc_d)$

69 *endFor*

70 *endIf*

71 *endIf*

72 *endIf*

73 *endIf*

74 *return constraint*

If RCA is *first-applicable*, then all the rules after current rule under consideration with the same effect as that of the current rule under consideration should evaluate to *N/A*. If RCA is *permit-unless-deny*, then all the rules after i^{th} rule with deny effect should be *N/A*. If RCA is *deny-unless-permit*, all the rules after i^{th} rule with permit effect should be *N/A*. If RCA is *deny-overrides* (or *permit-overrides*), then we only falsify deny (or permit) rules

with common attributes in rule body of current rule under consideration and mark the deny (or permit) rules with no common attribute for later to be used for propagation constraint. If i[th] rule is deny (or permit), then propagation constraint is satisfied, and we can return the constraint otherwise if it is permit (or deny) and mutation operator is not CRE (for CRE the later propagation constraint i.e step 31- 46 is not required) *and dominantRuleCollection* is not empty, we need to check for each rule in *dominantRuleCollection* whether falsifying this deny (or permit) rule will fire another permit (or deny) rule. If this is the case, we cannot falsify the current rule in *dominantRuleCollection.* Hence, we make it evaluate indeterminate and set *dominantIndeterminateFlag* to *true*. If any deny (or permit) rule evaluate to indeterminate, then for fault detection, all the permit(deny) rules should evaluate to *N/A*.

4.4 FDC Constraint and Tests Generation for RTT

Algorithm 5*: generateTestsForRTT(P)*

Input: Policy $P = \langle PT, RCA, [r_1, r_2, \ldots, r_n] \rangle$

Output: A set of access requests Q

1 $Q \leftarrow \epsilon$

2 *dominantIndeterminateFlag* ← *false*

3 *dominantRuleCollection* ← *null*

4 *constraint* ← *PT*

5 *For each Rule* r_i *in* $[r_1, r_2, \ldots, r_n]$ *in P, do*

6 *ruleConstraint* ← *constraint*

7 *ruleConstraint* ← *ruleConstraint* ∧ *ruleReachability(P,* r_i*)*

8 *ruleConstraint* ← *ruleConstraint* ∧ ¬ rt_i ∧ (rc_i ∨ *Error(*rc_i*))*

9 ruleConstraint ← *ruleConstraint* ∧ *rulePropagation(P,* r_i*,RTT)*

10 Q ← *Q U Z3-request(ruleConstraint)*

11 EndFor

12 return Q

Like CRE, we construct the FDC to kill the RTT by combining reachability, necessity and propagation constraint. The necessity constraint for RTT is that target of the rule under consideration should be *N/A* and rule condition should be *true*.

4.5 FDC constraint and tests generation for RTF

Algorithm 6*: generateTestsForRTF(P)*

Input: Policy P= < PT, RCA, $[r_1, r_2, \ldots, r_n]$*>*

Output: A set of access requests Q

1 Q ← ϵ

2 dominantIndeterminateFlag ← *false*

3 dominantRuleCollection ← *null*

4 constraint ← *PT*

5 For each Rule r_i *in* $[r_1, r_2, \ldots, r_n]$ *in P, do*

6 ruleConstraint ← *constraint*

7 ruleConstraint ← *ruleConstraint* ∧ *ruleReachability(P,* r_i*)*

8 ruleConstraint ← *ruleConstraint* ∧ $(rt_i \wedge rc_i) \vee$ *Error(*rb_i*)*

9 ruleConstraint ← *ruleConstraint* ∧ *rulePropagation(P,* r_i*,RTF)*

10 Q ← *Q U Z3-request(ruleConstraint)*

11 EndFor

12 return Q

The input to the algorithm is policy *P* = < *PT, RCA,* [r_1, r_2,..., r_n]> and output is set of generated access request *Q*. Initially we set *Q* to be empty (line 1). We set the *dominantIndeterminateFlag* to *false* and *dominantRuleCollection* to *null* to keep track of whether any dominant rule evaluated to indeterminate or not. If *RCA* is *first-applicable*, then any rule is dominant. If *RCA* is *permit-overrides* or *deny-unless-permit*, then the rules with permit effect will be the dominant rule. Similarly, rules with *deny* effect will be dominant in policy with *deny-overrides* or *permit-unless-deny* RCA. We then set the constraint expression from policy target if it is present, otherwise, the constraint is empty (line 4). The next step is to iterate over each rule in the policy (line 5) and construct sufficient condition constraint to kill the mutant. For each rule, the rule constraint is set to constraint expression (line 6) from policy target constraint in line 2. Then, for each rule in the policy, the rule constraint expression is concatenated with reachability constraint (line 7). Once the reachability constraint is met, the next task is to concatenate the rule constraint with necessity constraint for the fault type (line 8) and then concatenate the propagation constraint (line 9). The constructed rule constraint is the constraint with sufficient condition to kill the RTF mutant which is supplied to constraint solver (line 10) to obtain the value for the test inputs. When all the rules are processed, the generated set of test input - Q is returned.

4.6 FDC Constraint and Test Generation for RCT

Algorithm 7: *generateTestsForRCT(P)*

Input: Policy P= < *PT, RCA,* [r_1, r_2,..., r_n]>

Output: A set of access requests Q

1 $Q \leftarrow \epsilon$

2 *constraint* $\leftarrow$ *PT*

3 *For each Rule* r_i *in* $[r_1, r_2, \ldots, r_n]$ *in P, do*

4 *ruleConstraint* $\leftarrow \epsilon$

5 *ruleConstraint* $\leftarrow$ *constraint* $\wedge$ *ruleReachability(P,*r_i*)*

5 *ruleConstraint* $\leftarrow$ *ruleConstraint* $\wedge (rt_i \vee Error(rt_i)) \wedge \neg rc_i$

6 *ruleConstraint* $\leftarrow$ *ruleConstraint* $\wedge$ *rulePropagation(P,* r_i*, RCT)*

7 *Q* $\leftarrow$ *Q U Z3-request(ruleConstraint)*

8 *endFor*

9 *return Q*

The sufficient condition to kill the RCT mutants is same as that for CRE except for that condition of the rule under consideration should be *N/A*.

4.7 FDC Constraint and Test Generation for RCF

RCF and RTF have common sufficient condition for fault detection. Hence, Algorithm 6 for RTF is enough, and we don't need to consider RCF if RTF is in the fault model.

4.8 FDC Constraint and Test Generation for ANF

RCF and RTF have common sufficient condition for fault detection. Hence, Algorithm 6 for CRE is enough and we don't need to consider ANF if CRE is in the fault model.

4.9 FDC Constraint and Test Generation for RNF

RNF and RTF have similar sufficient condition to kill them. Hence, the constraint for RTF is enough and we don't need to consider RNF if RTF is in the fault model.

4.10 FDC Constraint and Test Generation for RER

RER and RTF have similar sufficient condition to kill them. Hence, the constraint for RTF is enough and we don't need to consider RER if RTF is in the fault model.

4.11 FDC Constraint and Test Generation for PTT

Algorithm 10*: generateTestsForPTT(P)*

Input: Policy P= < PT, RCA, [r_1, r_2, …, r_n]>

Output: A set of access requests Q

1 $Q \leftarrow \epsilon$

2 $constraint \leftarrow \neg PT \wedge (rt_1 \wedge rc_1)$

3 $Q \leftarrow Q$ *U Z3-request(constraint)*

4 *Return Q*

The sufficient condition to kill PTT mutant is to make policy target evaluate to *N/A* and make any rule's (say the first rule) target and condition evaluate to *true*.

4.12 FDC Constraint and Test Generation for PTF

Algorithm 11*: generateTestsForPTF(P)*

Input: Policy P= < PT, RCA, [r_1, r_2, …, r_n]>

Output: A set of access requests Q

1 $Q \leftarrow \epsilon$

2 $constraint \leftarrow PT \wedge (rt_1 \wedge rc_1)$

3 $Q \leftarrow Q$ *U Z3-request(constraint)*

4 *Return Q*

The sufficient condition to kill PTF mutant is to make policy target evaluate to *true* and make any rule's (say the first rule) target and condition evaluate to *true*.

4.13 FDC Constraint and Test Generation for FPR

Algorithm 8*: generateTestsForFPR(P)*

Input: Policy P= < PT, RCA, [r_1, r_2, ..., r_n]>

Output: A set of access requests Q

1 $Q \leftarrow \epsilon$

2 constraint $\leftarrow PT$

3 d $\leftarrow$ *find the position of first Deny Rule*

4 p $\leftarrow$ *find first Permit Rule after d*

5 *if such p and d does not exist*

6 *return*

7 constraint $\leftarrow rt_d \wedge rc_d$

8 constraint $\leftarrow rt_p \wedge rc_p$

9 *if RCA = first-applicable*

10 *For each rule r_i in [r_1, r_2, ..., r_{d-1}] in P, do*

11 constraint $\leftarrow$ constraint $\wedge \neg (rt_i \wedge rc_i)$

12 $Q \leftarrow Q$ *U Z3-request(constraint)*

13 *return Q*

The FPR mutants behave similarly to the original policy if RCA is other than *first-applicable*. Even for the first applicable, if there does not exist a pair of rules such that one is deny and other is permit, then the FPR mutant behaves the same as original policy. Hence, the sufficient condition to kill non-equivalent FPR mutant is to make target and condition of a pair of rules evaluate to *true* such that one of them is permit rule and the other is deny rule.

4.14 FDC constraint and test generation for FDR

***Algorithm 9**: generateTestsForFDR(P)*

Input: Policy P= < PT, RCA, [r_1, r_2,…, r_n]>

Output: A set of access requests Q

1 $Q \leftarrow \epsilon$

2 $constraint \leftarrow PT$

3 *p* $\leftarrow$ *find the position of first permit Rule*

4 *d* $\leftarrow$ *find first deny Rule after p*

5 *if such p and d does not exist*

6 *return*

7 $constraint \leftarrow rt_d \wedge rc_d$

8 $constraint \leftarrow rt_p \wedge rc_p$

9 *if RCA = first-applicable*

10 *For each rule r_i in [r_1, r_2,…, r_{d-1}] in P, do*

11 $constraint \leftarrow constraint \wedge \neg (rt_i \wedge rc_i)$

12 $Q \leftarrow Q$ *U Z3-request(constraint)*

13 *return Q*

The FDR mutants behave similarly to the original policy if RCA is other than *first-applicable*. Even for the first applicable, if there does not exist a pair of rules such that one is deny and other is permit, then the FDR mutant behaves the same as original policy. Hence, the sufficient condition to kill non-equivalent FDR mutant is to make target and condition of a pair of the rules evaluate to *true* such that one of them is permit rule and the other is deny rule.

4.15 FDC Constraint and Test Generation for RPTE

Algorithm 12: *generateTestsForRPTE(P)*

Input: Policy P= < PT, RCA, [r_1, r_2,…, r_n]>

Output: A set of access requests Q

1 $Q \leftarrow \epsilon$

2 $constraint \leftarrow PT$

3 *For each Rule* r_i *in* [r_1, r_2,…, r_n] *in P, do*

4 $ruleConstraint \leftarrow \epsilon$

5 $ruleConstraint \leftarrow constraint \wedge ruleReachability(P,r_i)$

6 $ruleConstraint \leftarrow ruleConstraint \wedge (rc_i \vee Error(rc_i))$

7 $ruleConstraint \leftarrow ruleConstraint \wedge rulePropagation(P,r_i)$

8 *For each* $AnyOf_j$ *in* [$AnyOf_{i1}$, $AnyOf_{ij}$,…, $AnyOf_{im}$] *in* rt_i, *do*

9 $anyConstraint \leftarrow ruleConstraint \wedge AnyOf_{i1} \wedge \neg AnyOf_{ij} \wedge AnyOf_{in}$

10 $Q \leftarrow Q \cup Z3\text{-}request(anyConstraint)$

11 *For each* $AllOf_k$ *in* [$AllOf_{ij1}$, $AllOf_{ijk}$,…, $AllOf_{ijl}$] *in* rt_i, *do*

12 $allConstraint \leftarrow ruleConstraint \wedge AnyOf_{i1} \wedge (\neg AllOf_{ij1} V AllOf_{ijk}$

$,\ldots \vee \neg AllOf_{ijl}) \wedge AnyOf_{in}$

13 $Q \leftarrow Q \cup Z3\text{-}request(anyConstraint)$

14 *return Q*

If there are *m AnyOf* clauses, then there should be m+1 expression such that one makes all of them *true* and the rest makes each of them evaluate to *false* while the other evaluates to *true*. In addition, for each *AnyOf* clause, if there are n *AllOf* clauses, then there should be n+1 expression such that one makes all of them evaluate to *false* and the rest makes each of them evaluate to *true* while others evaluate to *false*.

4.16 FDC constraint and test generation for CRC

Algorithm 13: *generateTestsForCCA(P)*

Input: Policy P= < PT, RCA, [r_1, r_2,…, r_n]>

Output: A set of access requests Q

1 *Q ← ϵ*

2 *Constraint1 ← PT*

3 *Constraint2 ← PT*

4 *p ← find the position of permit Rule and deny Rule such that at least one of them evaluates to true and other evaluates to true or error*

5 *d ← find first deny Rule after p*

6 *if such p and d does not exist*

7 *go to step 11*

8 *Constraint1 ← Constraint1* $\wedge$ $rt_d \wedge rc_d$

9 *Constraint1 ← Constraint1* $\wedge$ *Error*($rt_p \wedge rc_p$)

10 *Q ← Q U Z3-request(Constraint1)*

11 *Constraint2 ← Constraint2* $\wedge$ *Error for all the rules [r_1 … r_n]*

12 *Q ← Q U Z3-request(Constraint2)*

13 *Return Q*

The sufficient condition to kill CCA mutants is to make target and condition of a pair of the rules evaluate to *true* such that one of them is permit rule and other is deny rule. In addition, all the rules that can be made evaluated to an error should be evaluated to an error.

CHAPTER 5

Quantitative Analysis

This section presents our experiment that aims to evaluate the cost-effectiveness of the afore-mentioned testing methods by comparing them to *SMT*. As mutation score is commonly used as the main indicator of effectiveness for a testing method, we measure cost-effectiveness by comparing mutation score with test suite size i.e how many mutants are killed given the test suite size. Further, the test suite size reflects the average time of test execution. We will also discuss test generation time to reflect the run-time efficiency of testing methods.

5.1 Experiment Setup

The mutation-based test generation and all coverage-based test generation methods discussed in this work are implemented in open source tool - XPA (XACML Policy Analyzer) [15] which is based on Balana [13] - an open source implementation of XACML 3.0. Our experiment was performed on a 64bit Ubuntu laptop with 8th Generation Intel® Core™ i7-8550U Processor (1.80GHz 8MB) and 16.0GB DDR4 2400MHz. The experiments use various XACML 3.0 policies with different levels of complexity as shown in Table 5.1. K-Market is the sample policy from the Balana [13]. Sample, Sample-fa, and Sample-dup policies are created by us for this research to cover the language feature of XACML policies not covered by other policies in the literature. All other policies are from the literature. iTrust-X and fedora-rule3-X are synthesized from iTrust and fedora policies

respectively to study behavior in larger sized policies. The number of the rules ranges from 3 to 1,280 as depicted in Table 5.1 where '#' represents 'number of' i.e. '#Rules' means 'number of the rules' and LOC represents the line of code (markup) in the corresponding policies.

The experiment involves generating mutants of each policy by using the mutation operators in Table 3.1. Each mutation operator may generate several mutants for a given policy. For example, given a policy with *n* rules, CRE (Change Rule Effect) creates *n* mutants because it creates a mutant by changing the effect of each rule. The mutation operators in Table 3.1 is based on the operators in the mutant generator, XACMUT [4]. The next activity is to identify the number of equivalent mutants and non-equivalent mutants. As we applied strong mutation to generate SMT test suite, the number of mutants killed from the strong mutation-based test suite (SMT) are non-equivalent mutants while live mutants are equivalent mutants. Hence, we generate SMT test suite and run against the mutants to identify the equivalent and non-equivalent mutants. As a result, the analysis excludes mutants that are equivalent to their original policy. For example, the rule combining algorithms *permit-overrides* and *deny-overrides* make no difference with respect to a policy with permit-only (or deny-only) rules. Hence, a policy which only has permit rules and its RCA changed from *permit-overrides* to *deny-overrides* are equivalent. Our prior work on the formalization of semantic differences between combining algorithms [26] provides descriptions about the conditions under which two combining algorithms are equivalent. Table 5.1 lists the number of mutants for each subject policy.

As a next step, we generate test suites for each of the coverage-based test generation methods for each of the subject policies and run that test suite against each policy and

record the actual response of each test. We know each recorded response for each request (test case) is the correct response from the policy for this test case. Hence, it can be used as the Oracle value of this test case when the mutants are tested later. Finally, we run the test suite of each test generation method against mutants. Since mutants represent the faults that are likely to occur in XACML policies, mutation score is considered the indicator of the fault-detection capability, as commonly used by the software testing community [16].

Table 5.1: Policies used for the experiment

No	Subject Policy	LOC	#Rules	#Equivalent Mutants	#Non-equivalent Mutants
1	Conference [21]	228	15	1	91
2	fedora-rule3[1]	226	12	1	87
3	K-Market-blue [13]	84	4	1	27
4	K-Market-sliver [13]	58	3	1	21
5	K-Market-gold [13]	106	5	1	32
6	Sample	152	6	1	55
7	Sample-fa	114	4	0	42
8	Sample-dup	80	4	1	33

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

9	fedora-rule3-2	588	32	1	207
10	fedora-rule3-3	2748	212	1	927
11	iTrust[2]	1282	64	4	450
12	iTrust-5 [15]	6402	320	4	2242
13	iTrust-10 [15]	12806	640	4	4482
14	iTrust-20 [15]	25602	1280	4	8962

The following section presents the results of the conducted experiment.

5.2 Results

5.2.1 Test Suite Size and Test Generation Time

Table 5.2 presents the number of tests generated for each subject policy for each testing method. Typically, the test suite for NO-SMT has more tests. The test suite size of MC/DC and SMT are nearly the same however SMT test suite is not larger than MC/DC test suite for all the subject policies. The size of each *RC* test suite is the number of the rule in the policy and it is always the smallest. The difference in test suite size between NE-DC and DC as well as NE-MC/DC and MC/DC represents the *error* tests. Hence, there are only a few *error* tests. For fedora with 12 rules has six *error* tests, while that of iTrust3-5 has only one *error* tests for the policy with 320 rules. The reason is that most of the rules in the iTrustX policy are defined over the same set of attributes - when a test makes one rule evaluate to indeterminate, it will do the same for all other similar rules. As mentioned before, SMT is computationally costly because of the involved optimization. It took *SMT*

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

36.02 hours (129655675 ms in Table 5.3) to complete the test generation for *iTrust3-10*. The estimated *SMT* test generation time for *iTrust3-20* is nearly 10 days. As a result, we decided not to run SMT for iTrust-20. However, from *NO-SMT*, we know that for each mutant, there is at least one test that kills it.

Table 5.2: Number of tests generated

Subject Policy	RC	NE-DC	DC	NE-MC/DC	MC/DC	SMT	NO-SMT
coference3	15	15	16	24	25	25	78
fedora-rule3	12	19	25	24	30	23	62
kMarket-blue-policy	4	7	10	8	11	7	21
kMarket-gold-policy	3	6	9	6	9	5	16
kMarket-sliver-policy	5	9	13	9	13	8	23
sample	6	13	19	15	21	16	51
sample-fa	4	9	15	10	16	11	30
sample-dup	4	7	10	9	12	8	23
fedora-rule3-2	32	39	45	64	70	62	162
fedora-rule3-3	212	219	225	244	250	242	702
iTrust3	64	65	66	196	197	196	387

iTrust3-5	320	321	322	982	983	982	1923
iTrust3-10	640	641	642	1964	1965	1964	3843
iTrust3-20	1280	1281	1282	3928	3929	-	7683

Table 5.3 shows the test generation time for each of the method for each of the subject policies. The *RC* takes minimal test generation time as it has lesser number of tests. The test generation time is roughly proportional to test suite size except for the *SMT* because of the involved optimization of a test suite. All the testing methods are scalable except for the *SMT*. As test generation time for iTrust3-10 is nearly one and half days, we decided not to run *SMT* for *iTrust3-20* because approximated time is nearly 10 days and this shows that *SMT* is not scalable because of the involved mutation analysis for optimization of a test suite size. However, for a policy with roughly hundred or lesser rules, SMT test generation time is comparable to that of other testing methods.

Table 5.3: Test generation time (in milliseconds)

Subject Policy	RC	NE-DC	DC	NE-MC/DC	MC/DC	SMT	NO-SMT
conference3	488	438	408	689	716	2847	2019
fedora-rule3	328	548	686	652	536	2222	1628
kMarket-blue-policy	96	176	252	210	307	467	519
kMarket-gold-policy	71	154	221	150	221	343	385
kMarket-sliver-policy	130	252	328	231	335	704	626
sample	174	358	506	435	557	1835	1309
sample-fa	101	241	569	276	488	862	763
sample-dup	117	201	270	237	323	669	593
fedora-rule3-2	847	1062	1290	1790	1935	14853	4534
fedora-rule3-3	5368	7779	8120	9467	10304	930226	22625
iTrust3	1644	1555	3511	5923	8349	171716	10363
iTrust3-5	8711	8884	24415	49093	64749	15875846	48091

iTrust3-10	13890	19089	84304	146987	211230	129655675	176758
iTrust3-20	45908	422653	562857	499790	696428	*	709927

The next section presents the mutation score and summarizes whether the mutants type (based on mutation operators) are killed or not by each testing methods.

5.2.2 Fault Detection Capability

Table 5.4 presents the mutation score for each testing method for each subject policy. The mutation score for *SMT* and *NO-SMT* are 100% for all the policies, so we did not include them in Table 5.4. It should be noted that we did not generate and run *SMT* test suite for iTrust3-20. However, from *NO-SMT*, we know that for each mutant, there is at least one test that kills it. So, it will be 100% as well.

Table 5.4: Mutation Scores

Subject Policy	RC	NE-DC	DC	NE-MC/DC	MC/DC
conference3	75.82	75.82	78.02	97.8	100
fedora-rule3	64.37	85.06	85.06	88.51	88.51
kMarket-blue-policy	81.48	96.3	96.3	100	100
kMarket-gold-policy	80.95	95.24	95.24	95.24	95.24
kMarket-sliver-policy	81.25	100	100	100	100
sample	69.09	87.27	90.91	92.73	96.36
sample-fa	66.67	85.71	90.48	92.86	97.62
sample-dup	48.48	78.79	84.85	90.91	96.97
fedora-rule3-2	36.71	55.07	55.07	56.52	56.52
fedora-rule3-3	27.62	51.13	51.13	51.46	51.46
iTrust3	42.67	58.22	58.22	100	100
iTrust3-5	42.82	57.36	57.36	100	100

iTrust3-10	42.83	57.25	57.25	100	100
iTrust3-20	42.84	57.19	57.19	100	100

Table 5.5 presents the type of mutants that could not be killed by each testing method. In Table 5.5, the results for three kMarket policies are combined into one. Further, iTrust3 and fedora-rule3-2 can represent other policies in the group of iTrust-X and fedora-rule3-X from Table 5.1, hence only iTrust3 and fedora-rule3-2 are included. The absence of mutation operator in Table 5.5 represents that all the mutants of the corresponding type are killed by that method for that policy or the mutation operator is not applicable to the policy.

The mutation operator without asterisk implies all the mutants of the type are live. For example, RTT specifies, all the mutants of type RTT are not killed. The mutation operator with an asterisk means some are not killed while some are not. Single asterisk implies less than 50% of the type of mutant are not killed and double asterisk implies half or more percentage of mutants of the type are not killed.

Table 5.5: Live Mutants

Subject Policy	**Testing Method**				
	RC	**NE-DC**	**DC**	**NE-MC/DC**	**MC-DC**
Sample	RTT ** RCT PTF RPTE ** CRC ** {FA, DUP, PUD}	RTT ** RPTE **	RTT * RPTE **	RTT **	RTT *
Sample-dup	CRE* RTT RTF* RCT RER* RPTE** CRC	CRE* RTF* RCT RER* RPTE**	CRE* RTF* RER* RPTE**	RCT RPTE*	RPTE*
Sample-fa	RTT** RCT** PTT FDR RPTE**	FDR RPTE**	FDR RPTE**	FDR	FDR

	CRC**{DO, ODO, PUD}	CRC*{DO, ODO }		CRC*{DO, ODO }	
Conference3	RPTE** CRC* {FA, DUP}	RPTE** CRC*{FA, DUP}	RPTE**	 CRC*{FA, DUP}	
fedora-rule3-2	RTT** RTF** RCT** RER** RPTE** CRC {FA, PUD}	 RTF** RER** RPTE**	 RTF** RER** RPTE**	 RTF** RER** RPTE**	 RTF** RER** RPTE**
Kmarket	RTT** RCT* PTT RPTE** CRC** {FA, PUD}	RTT* RPTE**	RTT* RPTE**	RTT*	RTT*
iTrust	RTT CRC RPTE	 RPTE**	 RPTE**		

The mutation score increases for methods from left to right in Table 5.4 and live mutants type decreases as we go from left to right in Table 5.5. The mutation scores ranged from 27% to 81% for the *RC* tests and Table 5.5 shows that it could not kill a majority of the mutants. The mutation scores for *NE-DC, DC, MC/DC and NE-MC/DC* ranged from 51% to 100%. However, this does not mean that they have similar fault detection capability. It is clear from Table 5.4 that all the policies have greater (if not the same) *MC/DC* (or *NE-MC/DC)* mutation score than that from *DC* (or *NE-DC*). It is also supported from the result in Table 5.5 which implies that the difference between fault detection capability among *MC/DC* and *DC* is the ability of *MC/DC* to detect the RPTE mutant type.

It is also noteworthy that *error* tests (from *NE-DC* and *NE-MC/DC*) do not always necessarily contribute to fault detection. However, they are crucial for detecting faults resulted from CRC mutation operator as well as RCT mutation operators. Hence, the *error* version of the testing method is recommended for higher fault detection capability. Further, if we do not consider fedora-rule3-2 policy, Table 5.4 shows that the mutation score of *MC/DC* is above 90% in most of the policies. Further, Table 5.2 shows that *MC/DC* and *SMT* have comparable tests size and Table 5.3 shows that test generation time of *MC/DC* is much less than SMT. These data not necessarily but may lead to infer that *MC/DC* is runtime efficient and cost-effective test suite for achieving above 90% mutation score and provide high-quality assurance of XACML policies. However, we know from the fault detection condition that *MC/DC* does not explicitly satisfy the propagation constraint for fault detection. As a result, if we increase the size of the policy where reachability and necessity constraint does not necessarily make the propagation constraint to be *true* like in fedora-rule3, the mutation score of *MC/DC* dropped from 88% to 51%.

Hence, though *MC/DC* is likely to achieve good mutation score in many policies, it cannot assure that it will always lead to the high assurance of the XACML policies because it explicitly satisfies only the reachability and necessity constraint. However, it should also be mentioned that MC/DC satisfies the reachability and necessity constraint of the majority of the faults that could be introduced by mutation operators in Table 3.1. Hence, we could qualify *MC/DC* as near to weak mutation-based test generation method but that is not the case with *DC* and *RC.* In fact, RC is far from satisfying necessity constraint for many of the faults in Table 3.1.

5.2.3 Cost Effectiveness

While mutation score is a good indicator of the fault detection capability of a testing method, it does not account for cost-effectiveness of the test suite i.e the number of mutants killing capability of each test in a test suite. We consider the average number of Mutants Killed Per Test (MKPT) as the indicator for cost-effectiveness.

Table 5.6: MKPT Scores

Subject	RC	NE-DC	DC	NE-MC/DC	MC/DC	SMT	NO-SMT
conference3	4.6	4.6	4.44	3.71	3.64	3.64	1.17
fedora-rule3	4.67	3.89	2.96	3.21	2.57	3.78	1.4
kMarket-blue-policy	5.5	3.71	2.6	3.38	2.45	3.86	1.29
kMarket-gold-policy	5.67	3.33	2.22	3.33	2.22	4.2	1.31
kMarket-sliver-policy	5.2	3.56	2.46	3.56	2.46	4	1.39
sample	6.33	3.69	2.63	3.4	2.52	3.44	1.08
sample-fa	7	4	2.53	3.9	2.56	3.82	1.4
sample-dup	4	3.71	2.8	3.33	2.67	4.12	1.43
fedora-rule3-2	2.38	2.92	2.53	1.83	1.67	3.34	1.28

fedora-rule3-3	1.21	2.16	2.11	1.95	1.91	3.83	1.32
iTrust3	3	4.03	3.97	2.3	2.28	2.3	1.16
iTrust3-5	3	4.01	3.99	2.28	2.28	2.28	1.17
iTrust3-10	3	4.03	3.996	2.282	2.28	2.28	1.16
iTrust3-20	3	4.001	3.998	2.281	2.28	-	1.16

Table 5.6 shows MKPT scores for various testing methods discussed in this work. Further, it suggests that *RC* has good MKPT scores and in many policies, it even achieved the highest scores among other testing methods. However, it does not mean RC will have the best cost-effectiveness in all policies as it has lower MKPT scores in iTrust-X and fedora-rule3-X. *NO-SMT*, on the other hand, could achieve perfect mutation score but has lowest MKPT score in all the subject policies. *DC* (or *NE-DC*) has better MKPT than that of *MC/DC* (or *NE-MC/DC*). *SMT* and *MC/DC have* similar MKPT scores in many policies, however, *SMT* has either similar or better MKPT scores in all the subject policies than *MC/DC*. Hence, this shows that MC/DC is nearly as cost-effective as near-optimal test suite from *SMT* and so is the decision coverage. In fact, in some policies decision coverage (specifically *NE-DC*) has better MKPT score than *SMT* in many policies. However, they do not assure perfect mutation score in many policies and could not always provide high-quality assurance of the XACML policies.

5.3 Threats to Validity

There are many threats to validity. First, XPA is based on the Balana [13] – OASIS implementation of XACML and hence our results depend on the proper implementation of Balana. While analyzing some of the results when executing policy against tests, we have found some inconsistencies between XACML specification and results from the Balana. Hence, the *error* in Balana implementation could propagate into the experimental results that we presented in this chapter. Second, the subject policies do not necessarily represent all possible real-world XACML policies. Some of them originate from real XACML systems, others are demonstrating examples (e.g., kMarket) or synthesized (e.g., iTrustX, fedora-rule3-X). We have developed some sample policies (e.g., sample, sample-fa) to

capture the features not covered by policies in the literature of XACML. However, the subject policies used in this research might still have not covered all language features of XACML. Third, there is the possibility of different implementations for the coverage criteria. Our result is based on the implementation of coverage-based test generation methods in XPA. While we believe the proposed test generation algorithms reflect the essential fault detection capabilities of the corresponding coverage criteria, different test generation algorithms may lead to slightly different mutation scores and MKPT scores. Fourth, all testing methods use Z3 as the constraint solver. Z3 is currently one of the most efficient constraint solvers available. As different constraint solvers may result in different attribute values for the same constraint, using another constraint solver in the implementation may produce slightly different test suites and, thus, lead to slightly different mutation scores. Fifth, although the 14 mutation operators have represented a great variety of possible faults in XACML3.0 policies, they do not necessarily cover all possible faults in real-world XACML3.0 policies. In addition, several mutation operators in the literature [7] are not yet implemented in this work, including RUF (RemoveUniquenessFunction), AUF (AddUniquenessFunction), CNOF (Change-NOF-Function), and CLF (ChangeLogicalFunction). Nevertheless, the formalization of the fault detection conditions of incorrect rule targets and conditions has provided a foundation for dealing with these types of mutants.

CHAPTER 6

Conclusions

6.1 Summary

We have presented the approach to strong mutation testing of XACML3.0 policies and have formalized the strong fault detection conditions based on a comprehensive fault model of XACML 3.0 policies. On the one hand, these strong fault detection conditions have been exploited to generate near-optimal test suites of specific policies. On the other hand, the strong fault detection condition gave us insight into the fault detection capabilities of existing testing methods. It is clear from the fault detection condition that *MC/DC* is near to weak mutation-based test generation method because it satisfies the reachability and necessity constraint for most of the faults in the fault model. However, that is not the case with *DC* and *RC.* In fact, *RC* is far from satisfying necessity constraint for many of the faults. This suggests that though MC/DC is likely to achieve good mutation score in most of the policies, it still could not always assure high-quality of XACML policies as it does not explicitly satisfy the propagation constraint.

Further, the generation of the near-optimal test suite has enabled us to perform quantitative evaluations on the cost-effectiveness of the testing methods of XACML policies. The results from the quantitative section suggest that *MC/DC* is nearly as cost-effective as near-optimal test suite from *SMT*. *SMT* has perfect fault detection capability and good cost-effectiveness, however, it may be practically infeasible to apply for larger

policies with a larger number of the rules. The *MC/DC* including *DC* and *RC* is scalable and cost-effective as close to the optimal test suite, however, they could not always assure the high fault detection. Although *MC/DC* achieved good mutation score in most of the policies, it performed badly in some of the policies. As a result, if perfect fault detection capability is required and test suite size is not of prime concern, then we can use the *NO-SMT* test suite. Hence, it is the trade-off between fault detection capability, cost-effectiveness and test generation time.

The formulated fault detection conditions for fault model considered in this work is not limited to the existing mutation operators. They provide theoretical guidelines for developing new testing methods and dealing with faults created by new mutation operators.

6.2 Future work

In this work, we have determined the overall fault detection capability of *RC*, *DC* and *MC/DC* based on whether they satisfy the reachability and necessity constraints of all the mutation operators or not. However, this approach can be formally extended to qualitatively evaluate the fault detection capability of each of the testing method relating to fault type. The implication is that it neither needs to have a test suite nor the Oracle values.

One of the other future works is to look at opportunities for further optimizing the mutation-based test suite. Another research direction is to look for the opportunity to optimize NO-SMT test suite without involving costly mutation analysis for the optimization.

Finally, except the policies considered in this work, XPA would only be able to support the analysis of policies which have syntax like one in the policies used in this work.

However, XACML is a broad XML based specification which allows us to define various policies using various functions, syntax not involved in any of the policies in this work. As a result, extending the XPA to support various other functions and syntax in the policy specification to make it more robust XACML analyzer tool could be another work for the future.

REFERENCES

[1] Hu VC, Ferraiolo D, Kuhn R, Friedman AR, Lang AJ, Cogdell MM, Schnitzer A, Sandlin K, Miller R, Scarfone K. Guide to attribute-based access control (ABAC) definition and considerations (draft). NIST special publication. 2013 Apr;800(162).

[2] eXtensible Access Control Markup Language (XACML) Version 3.0. 22 January 2013. OASIS Standard., 2013. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[3] Martin E, Xie T. A fault model and mutation testing of access control policies. InProceedings of the 16th international conference on World Wide Web 2007 May 8 (pp. 667-676). ACM.

[4] Bertolino A, Daoudagh S, Lonetti F, Marchetti E. Xacmut: Xacml 2.0 mutants generator. InSoftware Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on 2013 Mar 18 (pp. 28-33). IEEE.

[5] Bertolino A, Lonetti F, Marchetti E. Systematic XACML request generation for testing purposes. InSoftware Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on 2010 Sep 1 (pp. 3-11). IEEE.

[6] Bertolino A, Daoudagh S, Lonetti F, Marchetti E. Automatic XACML requests generation for policy testing. InSoftware Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on 2012 Apr 17 (pp. 842-849). IEEE.

[7] Bertolino A, Daoudagh S, Lonetti F, Marchetti E. The X-CREATE Framework-A Comparison of XACML Policy Testing Strategies. InWEBIST 2012 Apr 18 (pp. 155-160).

[8] Bertolino A, Daoudagh S, Lonetti F, Marchetti E, Schilders L. Automated testing of extensible access control markup language-based access control systems. IET software. 2013 Aug 1;7(4):203-12.

[9] Bertolino A, Le Traon Y, Lonetti F, Marchetti E, Mouelhi T. Coverage-based test cases selection for XACML policies. InSoftware Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on 2014 Mar 31 (pp. 12-21). Ieee.

[10] Martin E, Xie T. Automated test generation for access control policies via change-impact analysis. InProceedings of the Third International Workshop on Software Engineering for Secure Systems 2007 May 20 (p. 5). IEEE Computer Society.

[11] Martin E, Hwang J, Xie T, Hu V. Assessing quality of policy properties in verification of access control policies. InComputer Security Applications Conference, 2008. ACSAC 2008. Annual 2008 Dec 8 (pp. 163-172). IEEE.

[12] Li Y, Li Y, Wang L, Chen G. Automatic XACML requests generation for testing access control policies. In SEKE 2014 Jul (pp. 217-222).

[13] Balana. (2012). Open source XACML 3.0 implementation. http://xacmlinfo.org/2012/08/16/balana-the-open-source-xacml-3-0-implementation/

[14] Martin E, Xie T. A fault model and mutation testing of access control policies. InProceedings of the 16th international conference on World Wide Web 2007 May 8 (pp. 667-676). ACM.

[15] XPA: XACML Policy Analyzer, Open source project at https://github.com/dianxiangxu/XPA

[16] Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing? InProceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering 2014 Nov 11 (pp. 654-665). ACM.

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

[18] DeMillo, R.A. and Offut, A. J. (1991). Constraint-based automatic test data generation, IEEE Trans. on Software Engineering, 17(9): 900-910, Sept. 1991.

[19] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE transactions on software engineering. 2011 Sep;37(5):649-78.

[20] Howden WE. Weak mutation testing and completeness of test sets. IEEE Transactions on Software Engineering. 1982 Jul(4):371-9.

[21] Fisler K, Krishnamurthi S, Meyerovich LA, Tschantz MC. Verification and change-impact analysis of access-control policies. InProceedings of the 27th international conference on Software engineering 2005 May 15 (pp. 196-205). ACM.

[22] Martin, E. and Xie, T. Automated test generation for access control policies, In Supplemental Proc. of ISSRE, November 2006.

[23] Hwang J, Xie T, Hu V, Altunay M. ACPT: A tool for modeling and verifying access control policies. InPolicies for Distributed Systems and Networks (POLICY), 2010 IEEE International Symposium on 2010 Jul 21 (pp. 40-43). IEEE.

[24] Hughes G, Bultan T. Automated verification of access control policies.

[25] Hughes G, Bultan T. Automated verification of access control policies using a sat solver. International journal on software tools for technology transfer. 2008 Dec 1;10(6):503-20.

[26] Xu D, Zhang Y, Shen N. Formalizing semantic differences between combining algorithms in XACML 3.0 policies. InSoftware Quality, Reliability and Security (QRS), 2015 IEEE International Conference on 2015 Aug 3 (pp. 163-172). IEEE.

[27] Xu D, Shrestha R, Shen N. Automated Coverage-Based Testing of XACML Policies. InProceedings of the 23nd ACM on Symposium on Access Control Models and Technologies 2018 Jun 7 (pp. 3-14). ACM.

[28] DeMilli RA, Offutt AJ. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering. 1991 Sep;17(9):900-10.

[29] DeMillo RA, Offutt AJ. Experimental results from an automatic test case generator. ACM Transactions on Software Engineering and Methodology (TOSEM). 1993 Apr 1;2(2):109-27.

[30] DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. Computer. 1978 Apr;11(4):34-41.

[31] Zheng Y, Zhang X, Ganesh V. Z3-str: a z3-based string solver for web application analysis. InProceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering 2013 Aug 18 (pp. 114-124). ACM.

[32] Offutt J. A mutation carol: Past, present and future. Information and Software Technology. 2011 Oct 1;53(10):1098-107.

[33] Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? InProceedings of the 27th international conference on Software engineering 2005 May 15 (pp. 402-411). ACM.