TOWARDS AUTOMATIC REPAIR OF XACML POLICIES

by

Shuai Peng

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2017

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Shuai Peng

Thesis Title: Towards Automatic Repair of XACML Policies

Date of Final Oral Examination: 01 March 2017

The following individuals read and discussed the thesis submitted by student Shuai Peng, and they evaluated his 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
Gaby Dagher, Ph.D	Member, Supervisory Committee
Jyh-haw Yeh, 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.

DEDICATION

I dedicate my thesis work to my parents, who have always loved me unconditionally and have taught me by example to work hard and to be nice to others.

ACKNOWLEDGEMENTS

Thank my committee members for their patience and precious time. A special thanks to Dr. Dianxiang Xu, who has spent numerous hours discussing with me about the problems I encountered during the research. Without his guidance, I could never have accomplished the research.

And I would like to thank my fellow graduate students, Milson Munakami and Nitish Dhakal. The discussions with you have been inspiring.

ABSTRACT

In a complex information system, controlling the access to resources is challenging. As a new generation of access control techniques, Attribute-Based Access Control (ABAC) can provide more flexible and fine-grained access control than Role-Based-Access Control (RBAC). XACML (eXtensible Access Control Markup Language) is an industrial standard for specifying ABAC policies. XACML policies tend to be complex because of the great variety of attribute types for fine-grained access control. This means that XACML policies are prone to errors and difficult to debug. This paper presents a first attempt at automating the debugging process of XACML policies. Two techniques are used for this purpose: fault localization and mutation-based policy repair. Fault localization produces an ordered list of suspicious policy elements by correlating the test results and the test coverage information. Mutation-based policy repair searches for potential fixes by mutating suspicious policy elements with predefined mutation operators. Empirical studies show that the proposed approach is able to repair various faulty XACML policies with one or two seeded faults. Among the scoring methods for fault localization that are studied in the experiment, Naish2 and CBI-Inc are the most efficient.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

RBAC	Role-Based Access Control
XACML	eXtensible Access Control Markup Language
OASIS	Organization for the Advancement of Structured Information Standards
SBFL	Spectrum-Based Fault Localization
PDP	Policy Decision Point
MC/DC	Multi-Condition/Decision Coverage
XPA	XACML Policy Analyzer
CRC	Change Rule Combining algorithm
CRE	Change Rule Effect
ANF	Add Negation Function
RNF	Remove Negation Function
RCF	Replace Comparison Function
CCF	Change Comparison Function
LOC	Line Of Code
XML	eXtensible Markup Language
XSLT	Extensible Stylesheet Language Transformations
UML	Unified Modeling Language
AOP	Aspect-Oriented Programming
CSV	Comma Separated Values

CHAPTER ONE: INTRODUCTION

Background

Access control is a mechanism for regulating user access to resources in an information system. As information systems are getting more and more complex, it is highly desirable to separate access control policies from system functionality since the functionality is subject to frequent changes. Another benefit of the separation is that access control policies can be changed on the fly without re-compiling.

Attribute-based access control (ABAC) grants or denies access based on various attributes of authorization elements[1], including 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). Thus ABAC offers a flexible and fine-grained access control.

XACML (eXtensible Access Control Markup Language) is a standard specification language for ABAC, which was proposed by OASIS [2]. XACML supports a variety of data types and functions for specifying attributes and their operations. While XACML policies allows for fine-grained access control, the complexity makes it more prone to faults caused by misunderstanding of requirements, coding errors during development and maintenance phase. Faults in XACML policies can cause serious consequences such as unauthorized accesses or denial of service.

Several methods have been proposed to generate test inputs (i.e., access requests) from a given XACML policy [3][4][5][6][7][8][9]. The expected responses (or oracle values) are determined by examining access control requirements. A policy’s actual response to a test input is compared with the oracle value. A test fails if the actual response is different from the oracle.

The debugging of XACML policies can happen in multiple phases in the system life cycle, including development, maintenance and operation. During the development phase, tests are created according to access control requirements. Before the deployment of the system, the tests are executed for verification and validation. During system maintenance, when security or functional requirements are changed, the XACML policy is run against the regression tests to find out broken tests or faults introduced by the changes. After the system is put into operation, there still might be residual faults as current testing and verification technologies cannot guarantee to eliminate all faults. When unexpected access control decisions are observed, we can find out the actual access requests that triggered the residual faults in the system logs. These access requests are similar to the tests in development and maintenance phases, and can be added to the regression test suite.

Test failure provides little insight in which element in the XACML policy is wrong. Actually testing is not concerned with finding out where the faults are and how to fix them. The activity of finding out faults and fixing them is often referred to as debugging. Similar to debugging a program, debugging an access control policy can be difficult and frustrating, especially when the access control logic is complex and the access control policy is large. Debugging an access control policy often relies on trial and error, especially when there are multiple faults. Thus a technique that automatically locates faulty elements in an XACML policy and fixes them is highly desirable.

Problem Definition

The policy repair problem is formulated as follows: given a faulty XACML policy (or policy set), along with a test suite where at least one test fails when executed against the policy (or policy set), make one or more changes to the faulty policy (or policy set) so that the revised policy will pass all the tests.

The above definition assumes an adequate test suite, which is not always true in reality. Intuitively, the more adequate the test suite is, the better the policy repair performs. As mentioned before, the test cases of an XACML policy may come from policy testing during system development and maintenance or from actual access requests in an operational system. Each test consists of test input (access request) and oracle value. The oracle value is used to determine whether the test passes or fails. An access request consists of attribute names, data types, and values.

It is possible that an attribute name, data type, or value in an access request is invalid. The response to such a request is typically NotApplicable or Indeterminate. In an operational system, access requests with invalid attribute names or values may come from malicious users who attempt to gain unauthorized access or render the system out of service. During the development and maintenance stages, policy testing should include not only normal test inputs but also tests with invalid attribute names and values.

In the empirical studies, for each subject policy, tests are generated automatically to achieve the Multi-Condition/Decision Coverage (MC/DC). MC/DC originated from NASA’s RTCA/DO-178B document, which requires level-A aviation software to achieve MC/DC of the software structure [15]. Prior works has applied MC/DC to automatic test generation of XACML policies in order to achieve high assurance of XACML policies

[16]. The MC/DC test suite of an XACML policy satisfies the MC/DC of each policy (or policy set) target, each rule target, and each rule condition. It also includes a test to make each policy (or policy set) target and each rule to evaluate to error (i.e., to cover the Indeterminate decisions).

Proposed Approach

The proposed approach to repair a faulty XACML policy includes two major steps: fault localization and mutation-based repair.

As shown in Figure 1.1, the proposed approach to policy repair is an iterative process because there might be multiple faults in the faulty policy. The approach first produces a list of suspicious policy elements by fault localization, and then generates mutants by applying mutation operators to the ranked suspicious elements, starting from the most suspicious one. If a mutant is a plausible fix, the process is repeated, otherwise the next mutant is tried. The repair fails when all mutants have been tried and none of them is a fix.

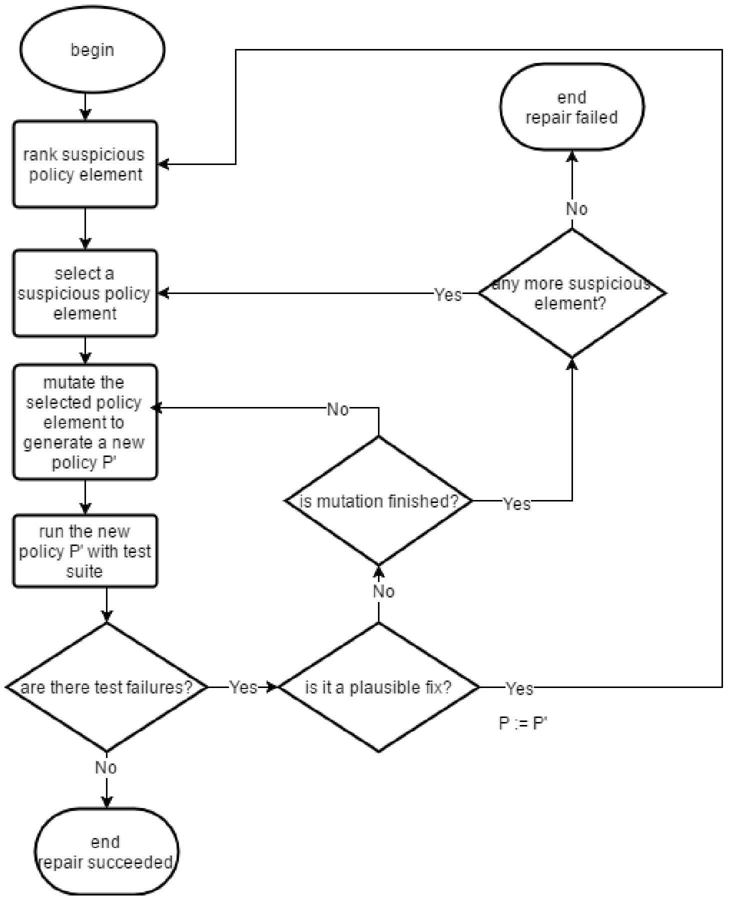

begin
rank suspicious policy element
select a suspicious policy element
mutate the selected policy element to generate a new policy P'
run the new policy P' with test suite
are there test failures?
No
end repair succeeded
Yes
is it a plausible fix?
Yes
P := P'
No
is mutation finished?
No
Yes
any more suspicious element?
Yes
No
end repair failed

Figure 1.1 Process of Automatic Repair

As illustrated in Figure 1.1, the first step of each iteration is ranking suspicious elements in a policy. This is accomplished by fault localization. As shown in Figure 1.2, a faulty policy P and a test suite are given as input. Each test in the test suites consists of an XACML request and the oracle value (expected decision). Given an XACML policy and an XACML request as input, XACML engine will output a decision. Test results are produced by comparing the actual decision and the oracle value. Meanwhile, coverage information is collected. Then a coverage matrix is built by combining the test results and coverage information. From the coverage matrix the suspicion score of each policy element can be calculated using a scoring method. Finally the policy elements are sorted by their corresponding suspicion scores, producing a sorted list of suspicious elements.

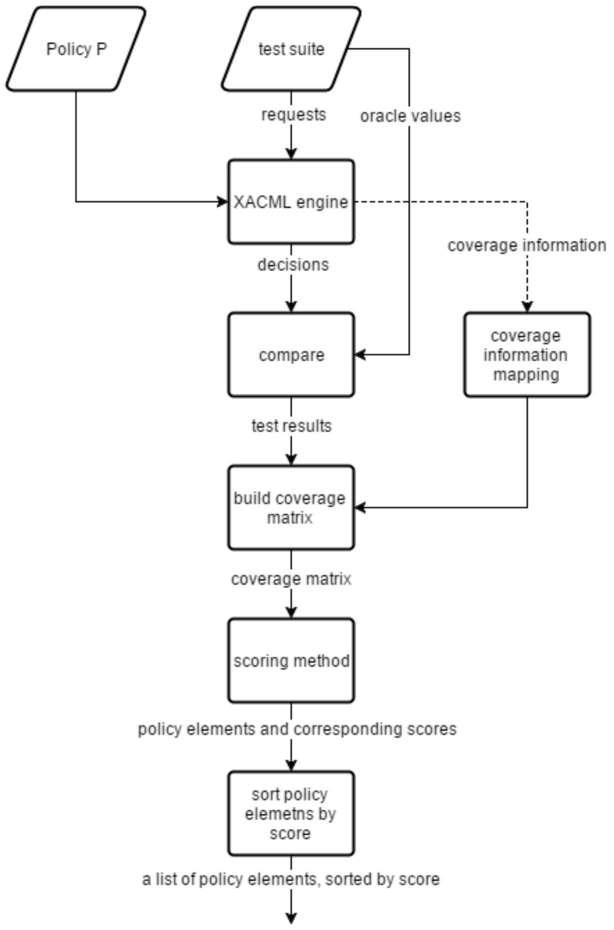


Figure 1.2 Process of Fault Localization

Chapter 4 will describe illustrate the fault localization, mutation based repair process in more details with a running example.

Outline

The remainder of this paper is organized as follows. Chapter 2 summarizes related work about the research topic. Chapter 3 gives an introduction to the structure of XACML policies, and illustrates it with an example. Chapter 4 describes the proposed approach with a running example then presents the general process. Chapter 5 elaborates on the implementation. Chapter 6 reports the empirical studies. Chapter 7 concludes this paper.

CHAPTER TWO: RELATED WORK

The existing work on policy debugging focuses on firewall policies. Marmorstein et al. used failed tests to locate faulty rules in a small firewall policy containing only several rules [10]. It does not identify faulty rules according to different fault types. Hwang et al. used failed tests to find the potential faulty rules based on structural coverage of firewall rules [11]. Two types of faults, wrong decisions and wrong predicates, were considered. In our approach, both passed and failed tests are used. Chen et al. proposed an approach to automatic correction of five types of faulty firewall rules: wrong order, missing rules, wrong predicates, wrong decisions, and wrong extra rules [12]. Part of this approach converts the given firewall rules into a firewall decision diagram (FDD) as a compact representation for reasoning about faulty rules. Compared to firewall rules, XACML policies are much more complex. Firewall rules are defined over a fixed set of network attributes and primarily specified in propositional logic, while rules in XACML policies are specified with predicates and functions with various data types.

Various fault localization techniques have been proposed for software debugging. This paper has adapted the scoring methods from spectrum-based fault localization (i.e., SBFL) for software debugging [13]. A program spectrum is an execution profile that indicates which parts of a program are active during a run. SBFL analyzes the differences in program spectra for passed and failed runs. Although the scoring methods in our approach are from SBFL, the variables are defined upon firing of policy elements, not coverage of policy elements.

Several testing methods have been proposed to generate test inputs from XACML policies [3] [4][5][6][7][8][9]. Testing is concerned with whether or not there are faults, whereas our work is concerned with how to locate and fix the faults. It is worth pointing out that the existing testing methods all use policy mutation to evaluate testing effectiveness. This paper, however, exploits mutation to fix faults. Our prior work has investigated coverage-based and firing-based approaches to fault localization of XACML policies [14]. It shows that firing-based fault localization outperforms coverage-based fault localization. Based on this result, this paper takes a step further to apply firing-based fault localization to rank policy elements for repair purposes.

CHAPTER THREE: XACML POLICIES

Structure of XACML Policies

The basic elements in XACML 3.0 language model are policy set, policy, rule, target, condition and combining algorithm [2]. Figure 3.1 shows the relationships between the main elements of XACML3.0.

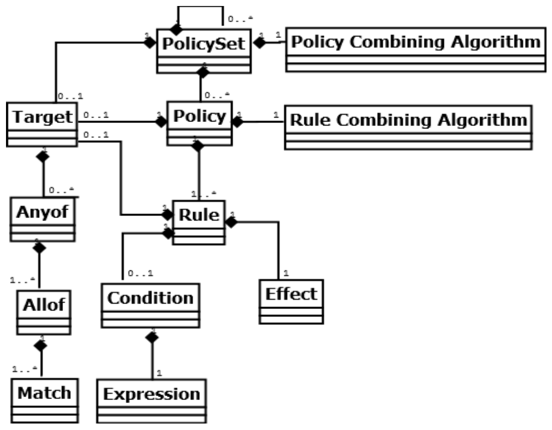


Figure 3.1 Language elements of XACML 3.0[14]

At the root of each XACML document is a policy or policy set. A policy or policy set defines the circumstances under which whether an access request should be granted. A policy set contains a target, a combining algorithm, and one or more policies or policy sets. The target decides if the XACML document is applicable to an access request. Policy combining algorithms include `deny-overrides`, `permit-overrides`, `deny-unless-permit`, `permit-unless-deny`, `first-applicable`, and `only-one-applicable`, etc. The combining algorithms combines the decisions of individual component policies or policy sets to form a final decision. For example, when the combining algorithm is `deny-overrides` and there is one component policy that evaluates to Deny, then the authorization decision of the whole policy set will be Deny, regardless of the decisions of other component policies.

Similarly, a policy contains a target, a combining algorithm, and one or more rules. A rule is the smallest unit of decision making. In addition to a target, a rule also contains an effect and a condition. The effect is either Permit or Deny. And the condition is a boolean expression which refines the applicability. As shown in Table 3.1, given an access request, the response of a rule can be Permit, Deny, NotApplicable or Indeterminate. The response is NotApplicable when the access request doesn't match with the target or condition of the rule. And the response is Indeterminate only when an error occurs during the evaluation.

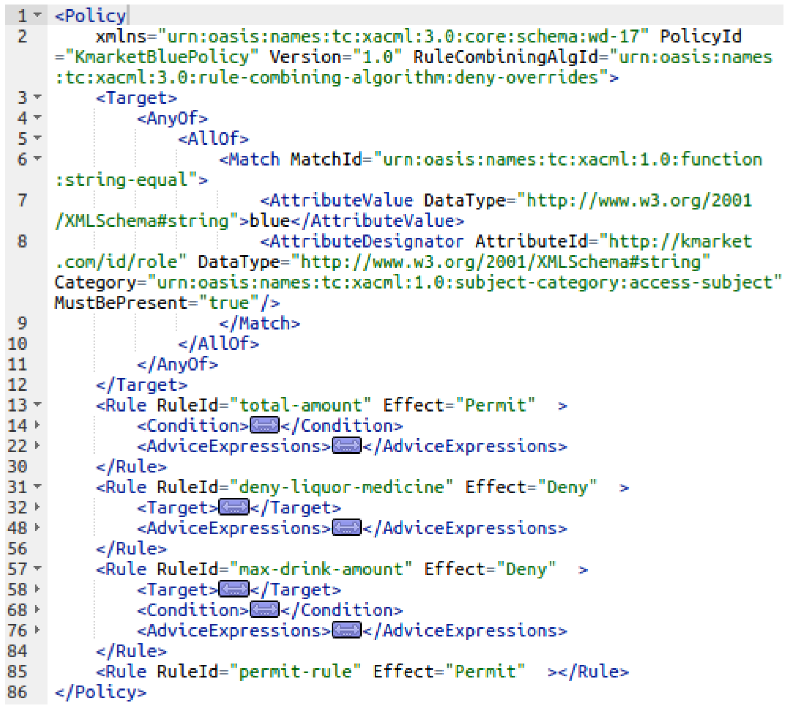
Table 3.1 Response of a Rule

target	condition	effect	response
false	-	-	NotApplicable
true	false	-	NotApplicable
true	true	Permit	Permit
true	true	Deny	Deny

In addition, a rule, policy, or policy set may have one or more obligation or advice expressions. This paper will not discuss about obligation and advice as they are irrelevant to the topic.

A Sample XACML Policy

Figure 3.2 shows an example XACML policy named KmarketBluePolicy (line 2).



```
1  <Policy
2      xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17" PolicyId
   ="KmarketBluePolicy" Version="1.0" RuleCombiningAlgId="urn:oasis:names
   :tc:xacml:3.0:rule-combining-algorithm:deny-overrides">
3      <Target>
4          <AnyOf>
5              <AllOf>
6                  <Match MatchId="urn:oasis:names:tc:xacml:1.0:function
   :string-equal">
7                      <AttributeValue DataType="http://www.w3.org/2001
   /XMLSchema#string">blue</AttributeValue>
8                      <AttributeDesignator AttributeId="http://kmarket
   .com/id/role" DataType="http://www.w3.org/2001/XMLSchema#string"
   Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
   MustBePresent="true"/>
9                  </Match>
10             </AllOf>
11         </AnyOf>
12     </Target>
13     <Rule RuleId="total-amount" Effect="Permit"  >
14         <Condition>▭</Condition>
22         <AdviceExpressions>▭</AdviceExpressions>
30     </Rule>
31     <Rule RuleId="deny-liquor-medicine" Effect="Deny"  >
32         <Target>▭</Target>
48         <AdviceExpressions>▭</AdviceExpressions>
56     </Rule>
57     <Rule RuleId="max-drink-amount" Effect="Deny"  >
58         <Target>▭</Target>
68         <Condition>▭</Condition>
76         <AdviceExpressions>▭</AdviceExpressions>
84     </Rule>
85     <Rule RuleId="permit-rule" Effect="Permit"  ></Rule>
86 </Policy>
```

Figure 3.2 A Sample XACML Policy

The rule combining algorithm is `deny-overrides` (line 2). The policy's target (lines 3-12) means `role="blue"`, where `role` is an attribute in the `subject` category and its type is `string`. There are four rules: `total-amount` (line 13), `deny-liquor-medicine` (line 31), `max-drink-amount` (line 57), and `permit-rule` (line 85). The policy target and rules are summarized in plain text in Table 3.2. The policy target (denoted as PT) is `role="blue"`. For rule `total-amount`, its effect is Permit, its target is `totalAmount>100`, and its condition is omitted. Its decision would be Permit if `totalAmount>100` evaluates to true. Similarly, rule `deny-liquor-medicine` would result in a Deny decision if `resource-id>Liquor ∨ resource-id=Medicine` evaluates to true. Rule `max-drink-amount` has both target and condition components. Its decision would be Deny if both its target and condition evaluate to true (i.e., `resource-id=Drink ∧ amount>10`).

Rule `permit-rule` has neither a target nor a condition. It results in a Permit decision whenever it is applied.

Table 3.2 Main Policy Elements in the Sample Policy

Policy Element	Target	Condition	Effect
Policy Target(PT)	role = Blue	-	
total-amount	totalAmount > 100	-	Permit
deny-liquor-medicine	Resource-id > “liquor” ∨ resource-id = “medicine”	-	Deny
max-drink-amount	Resource-id = “drink”	account > 10	Deny
permit-rule	-	-	Permit

Similar to the case of policy set, the authorization decision of a policy depends not only on the target and rules, but also on the rule combining algorithm. In this example, if the rule `deny-liquor-medicine` evaluates to Deny, the whole policy will evaluates to Deny, as the rule combining algorithm is `deny-overrides`. In such case, the remaining rules in the policy will be skipped since their decision won’t affect the overall decision.

Note that although Policy Decision Point (PDP) outputs only 3 kind of decisions: Permit, Deny and Indeterminate, internally Indeterminate is divided into 3 different decisions: Indeterminate {P}, Indeterminate {D} and Indeterminate {DP}. A rule or policy produces a Indeterminate {P} when an error occurred during evaluation, and the decision would be Permit if the error had not occurred. Similarly, a rule or policy produces a Indeterminate {D} when an error occurred during evaluation, and the decision would be Deny if the error had not occurred. An Indeterminate {DP} is produced when

Indeterminate {P} and Indeterminate {D} are combined. To make full use of coverage information from the PDP, all 6 kinds of decisions are used and a strict matching strategy is adopted, e.g. a test is deemed as failed if the oracle is Indeterminate {D} and the actual result is Indeterminate {P}.

CHAPTER FOUR: AUTOMATIC REPAIR OF XACML POLICIES

A Running Example

Consider the sample XACML policy in Chapter 3. It has two faults:

a. The effect of rule `total-amount` should be Deny, not Permit;

b. The target of rule `deny-liquor-medicine` should be `resource-id="Liquor" ∨ resource-id="Medicine"`, not `resource-id>"Liquor" ∨ resource-id="Medicine"`.

The original version of KmarketBluePolicy is one of the policies in a demonstration application of Balana [17], which is currently the only open-source implementation of XACML 3.0.

Table 4.1 shows the MC/DC test suite generated automatically from the correct version of KmarketBluePolicy by the open source XPA (XACML Policy Analyzer) tool[1]. The valid attribute names in access requests are `role`, `resource-id`, `amount`, and `totalAmount`. A test input may also consist of invalid attribute names and their attribute values. In Table 4.1, there are seven tests where all attribute names are valid. Each of the remaining tests (Test 1, 3, 5, 10) includes an invalid attribute name. The invalid attribute names are generated randomly to produce error conditions when the policy is tested. In this case, XPA simply uses Indeterminate as the attribute value, which indicates that an error occurrence is expected during policy testing.

[1] https://github.com/dianxiangxu/XPA. It includes the policies and test suites used in this paper.

Table 4.1 Test suite for the sample policy

No	***Input (attribute names and values in request)***					***Oracle Value***
	role	resource-id	amount	total Amount	invalid attribute, value	
1					nzocphnmz1, Indeterminate	NotApplicable
2	ak			0		NotApplicable
3	Blue			0	6m9dv7gdw6, Indeterminate	Indeterminate {DP}
4	Blue	k		0		Permit
5	Blue	Drink		0	j4yxpw95g1, Indeterminate	Indeterminate {DP}
6	Blue	Drink	0	0		Permit
7	Blue	Drink	11	0		Deny
8	Blue	Liquor		0		Deny
9	Blue	Medicine		0		Deny
10	Blue				o5eqqyvjdx, Indeterminate	Indeterminate {DP}
11	Blue			101		Deny

When the test suite in Table 4.1 is executed with the sample policy in Chapter 2, Test 5, 6, 8, and 11 shall fail. Consider Test 11, where `role="Blue"` and `totalAmount=101`, the oracle value is Deny. However, the actual response of the faulty policy is Permit because the effect of rule total-amount is Permit.

The goal is to repair the given KmarketBluePolicy policy such that the revised policy will pass all the tests in Table 4.1. Ideally, the revised policy will be identical to the original KmarketBluePolicy in Balana's Kmarket demonstration application.

The first step is to determine the policy elements (including the policy target, and individual rules, and the rule combining algorithm) that likely contain the faults. This is achieved by ranking all policy elements according to their suspicion scores obtained from the correlation between the firing information of policy elements and the test execution results (i.e., pass and fail). The higher the score, the more likely that the policy element is faulty.

A policy target (or policy set target) is said to be fired by a test if it is evaluated to true when the test is executed. A rule is said to be fired by a test if its target and condition are both evaluated to true when the test is executed. Note that the rule combining algorithm will always be evaluated. Therefore, the firing information of rule combining algorithm is not meaningful. Its suspicion score can be defined in different ways (e.g., highly suspicious or least suspicious). For simplicity, herein the rule combining algorithm is treated as the most suspicious element in a policy and attempts to change the rule combining algorithm first. The reason is that the number of repair attempts is small as there are only 11 rule combining algorithms.

Table 4.2 Firing of the policy target and rules in the sample policy

Test No	***Firing of policy target and rules***					***Test Result***
	PT	total-amount	deny-liquor-medicine	max-drink-amount	permit-rule	
1						Pass
2						Pass
3	x				x	Pass
4	x				x	Pass
5	x		x			Fail
6	x		x			Fail
7	x		x			Pass
8	x				x	Fail
9	x		x			Pass
10	x				x	Pass
11	x	x			x	Fail
$S_{tarantula}$	0.583	1.0	0.636	0.0	0.538	

Table 4.2 shows the firing information of the policy target and each rule in the sample policy when the test suite in Table 4.1 is executed, where ‘x’ means that the policy element in the given column is fired by the test in the given row. The set of failed

tests is {Test 5, Test 6, Test 8, Test 11}. To correlate policy elements with test results, the following four variables are associated with each policy element:

- a_{00}: number of passed tests in which the policy element was not fired
- a_{01}: number of failed tests in which the policy element was not fired
- a_{10}: number of passed tests in which the policy element was fired
- a_{11}: number of failed tests in which the policy element was fired

For each policy element, the subscript i in a_{ij} refers to whether the policy element is fired (i =1) or not (i =0), whereas j is concerned with the number of passed tests (j =0) or failed tests (j =1). For each policy element, a suspicion score is calculated by feeding a_{ij} to a scoring method and then sort all policy elements in the descending order of their scores. For example, the suspicion scores of the policy target and rules in Table 4.2 are 0.583, 1.0, 0.636, 0, and 0.538, respectively when the following Tarantula scoring method [18] is applied:

$$S_{Tarantula} = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}}+\frac{a_{10}}{a_{10}+a_{00}}}$$

The resultant ranking is <total-amount, deny-liquor-medicine, PT, permit-rule, max-drink-amount >. Then the rule combining algorithm (denoted as CA) is put at the beginning of the rankings. Therefore, the complete suspicion ranking of all policy elements is <CA, total-amount, deny-liquor-medicine, PT, permit-rule, max-drink-amount>.

Next step would be repairing the faulty policy according to the suspicion rankings of the policy elements. The faulty policy is repaired by changing the suspicious policy elements, starting from the most suspicious one. Herein the action of changing a policy is referred as policy mutation and a revised policy as a policy mutant. Policy mutation is performed by applying predefined mutation operators to the current policy. Mutation operators are defined with respect to the constructs of policy elements. Table 4.3 shows the mutation operators used in the current approach. Each operator aims at fixing a particular type of faults in a policy element.

Table 4.3 Mutation operators and target faults.

Operator	***Meaning***	***Fault to be fixed***
Change Rule Combining algorithm (CRC)	Replace the existing rule combining algorithm with another rule combining algorithm	Wrong rule combining algorithm
Change Rule Effect (CRE)	Change the rule effect by replacing permit with deny or deny with permit	Wrong rule effect
Add Negation Function (ANF)	add the not function as the first function of a rule condition element	Missing negation in a condition element
Remove Negation Function (RNF)	Remove the not function in a rule condition	Extra negation in a condition element
Replace Comparison Function (RCF)	Replace the comparison function in a target with a different one	Wrong comparison function in target
Change Comparison Function (CCF)	Change a comparison function in rule condition	Wrong comparison function in rule condition

In the above example, the rule combining algorithm is assumed to be the most suspicious element. So the mutation operator CRC (change rule combining algorithm) is applied to the faulty policy first. Consider changing the rule combining algorithm from `deny-overrides` to `permit-overrides`. The resultant policy mutant is denoted as CRC1. Run the test suite against CRC1 and CRC1 will fail the following set of tests {Test 3, Test 5, Test 7, Test 8, Test 9, Test 10, Test 11}. Apparently CRC1 is not a valid repair. As it is unknown how many faults are there in the policy, however, CRC1 might be a step toward the right direction or it might have introduced another fault. In the former case, the debugging process should be repeated until the faulty policy is repaired successfully. In the latter case, we would want to give it up and try other policy mutants.

Assuming that multiple faults in the same policy are independent, whether or not the mutation is in the right direction can be determined by examining the set of failed tests. The mutant is considered to be a plausible intermediate fix if the set of tests failed by the policy mutant is a subset of the tests failed by the faulty policy before the mutation. Apparently, CRC1 is not a plausible fix because it fails more tests than the example faulty policy does. As there are other candidate rule combining algorithms, we continue to create new CRC mutants and evaluate if any of these mutants is a plausible fix.

For KmarketBluePolicy, it turns out that none of the CRC mutants is a plausible fix. So now all possible attempts about the most suspicious policy element are completed. Now we move on to the next candidate element – the rule `total-amount`. Since a rule has several components (target, condition, and effect), there can be a number of applicable mutation operators. For simplicity, here we first consider the mutation operator CRE (change rule effect). The mutant after applying CRE to the rule total-amount is denoted as CRE1, where the rule effect is changed from permit to deny. Running the test suite against CRE1 will result in the following set of failed tests: {Test 5, Test 6, Test 8}, as shown in Table 4.4. This is a subset of the failed tests in Table 4.2. Thus, CRE1 is a plausible fix. Then we continue to apply the above debugging process to CRE1. Specifically, we create new suspicion rankings of the policy elements in CRE1 except for the rule combining algorithm because it is already shown non-promising as discussed before. According to Table 4.4, the suspicion rankings are <deny-liquor-medicine, PT, permit-rule, total-amount, max-drink-amount>.

Table 4.4 Suspicion scores of policy elements in CRE1

Test No	***Firing of policy elements***					***Test Result***
	PT	total-amount	deny-liquor-medicine	max-drink-amount	permit-rule	

1						Pass
2						Pass
3	x				x	Pass
4	x				x	Pass
5	x		x			Fail
6	x		x			Fail
7	x		x			Pass
8	x				x	Fail
9	x		x			Pass
10	x				x	Pass
11	x	x				Pass
$S_{tarantula}$	0.571	0.0	0.727	0.0	0.471	

Now the repair attempt focuses on the rule `deny-liquor-medicine` in CRE1. The applicable mutation operators are CRE (change rule effect) and RCF (replace comparison function in target). CRE is not promising. Instead, a RCF mutant that replaces the function “string-greater-than” to “string-equal” will pass all the tests. This mutant, named CRE1_RCF2_1, is a successful repair of the faulty policy KmarketBluePolicy. And it is identical to the original policy KmarketBluePolicy.

General Process for Automatic Policy Repair

Figure 1.1 shows the general process of automatic policy repair. The two key techniques of automatic policy repair are fault localization and mutation-based repair. Fault localization aims to rank all policy elements according to their suspicion scores obtained from the correlation between the firings of policy elements and the test results. In the running example, the Tarantula scoring method is used. In fact, there are various scoring methods. Table 4.5 summarizes the scoring methods implemented in the proposed approach. Tarantula [18] is one of the pioneer tools for software fault localization. Naish2 and CBI-Inc are among the best performing methods for software fault localization, whereas Sokal is among the average ones [13]. These representative methods are adapted for use in the fault localization of XACML policies and their performance are compared from the perspective of automatic policy repair.

Mutation-based repair applies mutation operators to each suspicious policy element in order to find an intermediate or final fix. If a policy mutant passes all the tests, it is a final fix. In this case, the faulty policy has been repaired successfully. If a policy mutant fails some tests that are a subset of the failed tests before the mutation, it is considered a plausible intermediate fix in the right direction. In this case, we continue to apply the debugging process to the policy mutant. However, if no fix is found for the current suspicious element after applying all mutation operators, the next suspicious policy element is selected for further mutation. If no fix is found after all policy elements have tried, then the faulty policy cannot be repaired by the approach.

Table 4.5 Scoring methods for fault localization

Method Name	***Formula***
CBI-Inc	$\frac{a_{11}}{a_{11}+a_{10}} - \frac{a_{11}+a_{01}}{a_{11}+a_{01}+a_{10}+a_{00}}$
Naish2	$a_{11} - \frac{a_{10}}{a_{10}+a_{00}+1}$
Sokal	$\frac{2(a_{11}+a_{00})}{2(a_{11}+a_{00})+a_{01}+a_{10}}$
Tarantula	$\frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}}+\frac{a_{10}}{a_{10}+a_{00}}}$

The proposed approach is not intended to repair all possible faulty policies automatically due to the theoretical and implementation challenges. Automatic policy repair essentially tries to search all possible mutants of the faulty policy. For a policy with n faults, the mutation operators may be applied to each policy element for up to n

times. The number of possible mutants grows exponentially with the number of faults (i.e., the number of repetitions that mutation is applied). For a multi-fault policy with a large number of policy elements, it could be too slow to be of practical use when no fix can be found. The proposed approach can estimate remaining time, and allows using a predefined timeout to terminate the search. Nevertheless, as will be shown in the empirical studies, our approach can repair faulty XACML policies with one or two seeded faults.

CHAPTER FIVE: IMPLEMENTATION

This chapter presents how the proposed approach to policy repair is implemented. The program architecture consists of four major components: `Coverage Analysis`, `Fault Localization`, `Mutation`, and `Repairing`.

Coverage Analysis

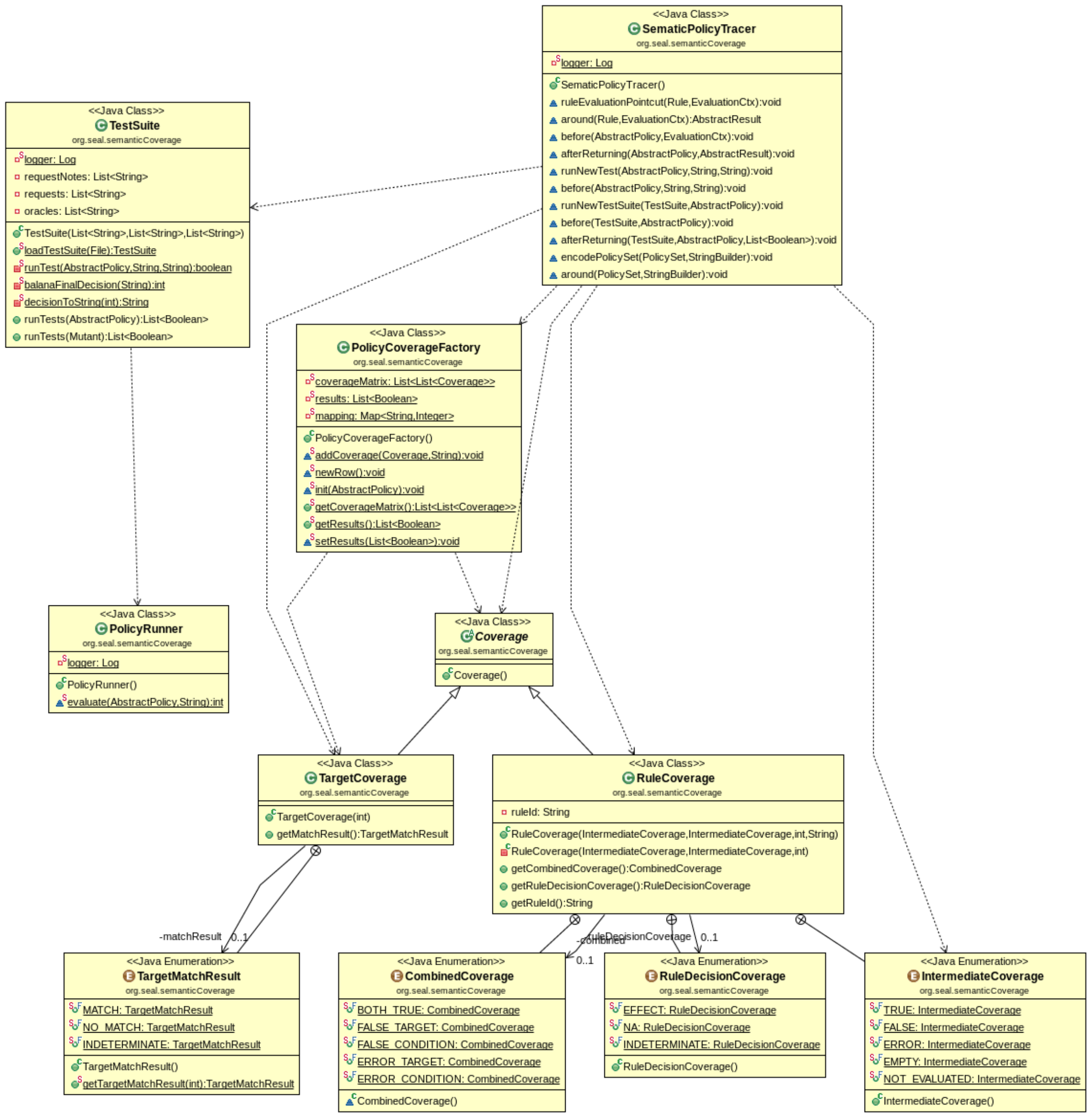
The component `Coverage Analysis` deals with running tests and collecting coverage information. There are four modules in this component: `Coverage`, `TestSuite`, `PolicyCoverageFactory`, and `PolicyTracer`. Figure 5.1 shows the UML class diagram of this component.

<u>Coverage</u>

The coverage information of rules and targets are defined differently, so there are two kinds of coverage information: `RuleCoverage` and `TargetCoverage`. They are modeled as two classes and they both extend the abstract class `Coverage`, so that both kind of coverage information can be handled in a uniformed way.

There can be only three evaluation results of a target: `MATCH`, `NOT_MATCH` and `INDETERMINATE`. The evaluation result is `INDETERMINATE` when an error occurred during the evaluation. If no error occurred during the evaluation, and the target is evaluated to be true, the evaluation result is `MATCH`, otherwise it is `NOT_MATCH`. So evaluation result of a target is modeled as Enum type `TargetMatchResult`.

The coverage information of a rule is more complex as a rule has a target and a condition, the coverage of which must be both taken into account. As summarized in Table 3.1, when a rule is evaluated, the target is evaluated first, and the condition is evaluated only if the target is evaluated to be true.



<<Java Class>>
SematicPolicyTracer
org.seal.semanticCoverage
logger: Log
SematicPolicyTracer()
ruleEvaluationPointcut(Rule,EvaluationCtx):void
around(Rule,EvaluationCtx):AbstractResult
before(AbstractPolicy,EvaluationCtx):void
afterReturning(AbstractPolicy,AbstractResult):void
runNewTest(AbstractPolicy,String,String):void
before(AbstractPolicy,String,String):void
runNewTestSuite(TestSuite,AbstractPolicy):void
before(TestSuite,AbstractPolicy):void
afterReturning(TestSuite,AbstractPolicy,List<Boolean>):void
encodePolicySet(PolicySet,StringBuilder):void
around(PolicySet,StringBuilder):void
<<Java Class>>
TestSuite
org.seal.semanticCoverage
logger: Log
requestNotes: List<String>
requests: List<String>
oracles: List<String>
TestSuite(List<String>,List<String>,List<String>)
loadTestSuite(File):TestSuite
runTest(AbstractPolicy,String,String):boolean
balanaFinalDecision(String):int
decisionToString(int):String
runTests(AbstractPolicy):List<Boolean>
runTests(Mutant):List<Boolean>
<<Java Class>>
PolicyCoverageFactory
org.seal.semanticCoverage
coverageMatrix: List<List<Coverage>>
results: List<Boolean>
mapping: Map<String,Integer>
PolicyCoverageFactory()
addCoverage(Coverage,String):void
newRow():void
init(AbstractPolicy):void
getCoverageMatrix():List<List<Coverage>>
getResults():List<Boolean>
setResults(List<Boolean>):void
<<Java Class>>
PolicyRunner
org.seal.semanticCoverage
logger: Log
PolicyRunner()
evaluate(AbstractPolicy,String):int
<<Java Class>>
Coverage
org.seal.semanticCoverage
Coverage()
<<Java Class>>
TargetCoverage
org.seal.semanticCoverage
TargetCoverage(int)
getMatchResult():TargetMatchResult
<<Java Class>>
RuleCoverage
org.seal.semanticCoverage
ruleId: String
RuleCoverage(IntermediateCoverage,IntermediateCoverage,int,String)
RuleCoverage(IntermediateCoverage,IntermediateCoverage,int)
getCombinedCoverage():CombinedCoverage
getRuleDecisionCoverage():RuleDecisionCoverage
getRuleId():String
-matchResult
0..1
-combined
0..1
-ruleDecisionCoverage
0..1
<<Java Enumeration>>
TargetMatchResult
org.seal.semanticCoverage
MATCH: TargetMatchResult
NO_MATCH: TargetMatchResult
INDETERMINATE: TargetMatchResult
TargetMatchResult()
getTargetMatchResult(int):TargetMatchResult
<<Java Enumeration>>
CombinedCoverage
org.seal.semanticCoverage
BOTH_TRUE: CombinedCoverage
FALSE_TARGET: CombinedCoverage
FALSE_CONDITION: CombinedCoverage
ERROR_TARGET: CombinedCoverage
ERROR_CONDITION: CombinedCoverage
CombinedCoverage()
<<Java Enumeration>>
RuleDecisionCoverage
org.seal.semanticCoverage
EFFECT: RuleDecisionCoverage
NA: RuleDecisionCoverage
INDETERMINATE: RuleDecisionCoverage
RuleDecisionCoverage()
<<Java Enumeration>>
IntermediateCoverage
org.seal.semanticCoverage
TRUE: IntermediateCoverage
FALSE: IntermediateCoverage
ERROR: IntermediateCoverage
EMPTY: IntermediateCoverage
NOT_EVALUATED: IntermediateCoverage
IntermediateCoverage()

Figure 5.1 UML Class Diagram of Package Coverage

The evaluation results of an individual target and condition can be defined in a uniformed way: The result is `TRUE` if the target or condition is evaluated to be true;

`FALSE` if it is evaluated to be false; `ERROR` if an error occurred during evaluation. And there are two additional cases: `EMPTY` if the target or condition is an empty element, which can be regarded as a special case of `TRUE`; `NOT_EVALUATED` if the target or condition is not touched by the test. A target is not touch if the evaluation has finished before this target is reached. A condition is not touched if the result of the target is `FALSE` or `ERROR`. This uniformed representation of evaluation results is modeled as Enum type `IntermediateCoverage`.

The coverage information of a rule is a combination of that of the target and condition in that rule. Rule coverage can be defined in two ways, with different granularity.

One way is to define it in the same way as in Table 3.1, except that the result would be `INDETERMINATE` if an error occurred during the evaluation of either the target or the condition. This is modeled by Enum type `RuleDecisionCoverage`.

Another definition is more fine grained: the result is `FALSE_TARGET` if the target is evaluated to be false; `FALSE_CONDITON` if the condition is evaluated to be false; `ERROR_TARGET` if an error occurred during the evaluation of the target; `ERROR_CONDITON` if an error occurred during the evaluation of the condition; `BOTH_TRUE` if both are evaluated to be true. This is modeled in Enum type `CombinedCoverage`.

<u>TestSuite</u>

The module `TestSuite` models a test suite generated from an XACML policy. It consists of a list of XACML requests and a list of corresponding oracle values. Each pair of request and oracle value is a test. In addition, a list of request notes is defined in the class to ease debugging. A request note is used as an annotation of a request.

A test suite can be created from the code that generates test suites from XACML policies or loaded from hard drive. The second way allows users to load previously saved auto-generated tests, or to load manually created tests, which is handy for debugging purpose. A saved test suite consists of a CSV file and some XACML request files. In the CSV file, in the first column are file paths to requests, which are XML files, and in the second column are oracle values.

The `runTests()` method runs the test suite on an XACML policy, and it calls the private method `runTest()` to run a single test in the test suite. Both methods are instrumented by AspectJ to collect coverage information while the tests are running. `runTest()` in turn calls `PolicyRunner.evaluate()` to evaluate an XACML request against an XACML policy.

<u>PolicyCoverageFactory and XpathSolver</u>

The `PolicyCoverageFactory` module is a “global variable” to store coverage information. All the data and method members in this module are static.

`PolicyCoverageFactory` has a `coverageMatrix` to store a matrix of `Coverage` objects. The number of rows is equal to the number of tests in the `TestSuite`, and the number of columns is equal to the number of policy elements in the XACML policy. An example of coverageMatrix is the one in Table 4.2. `PolicyCoverageFactory` also has `results`, a list of booleans, to store test results. The size of results is equal to number of tests in the `TestSuite`.

`PolicyCoverageFactory.init()` is called when a test suite starts to run, and initialize the coverageMatrix with an empty list. The `newRow()` method is called when a test starts to run, and adds a new row in the `coverageMatrix`. The `addCoverage()` method is called when a policy element is touched by a test, and inserts a `Coverage` object to the last row in the `coverageMatrix`.

When adding the `Coverage` of a policy element to the `coverageMatrix`, we need to know which column to add the `Coverage` object to, so a mapping from each policy element to a column in the `coverageMatrix` is needed. And this is the purpose of the data member `mapping` in the `PolicyCoverageFactory`.

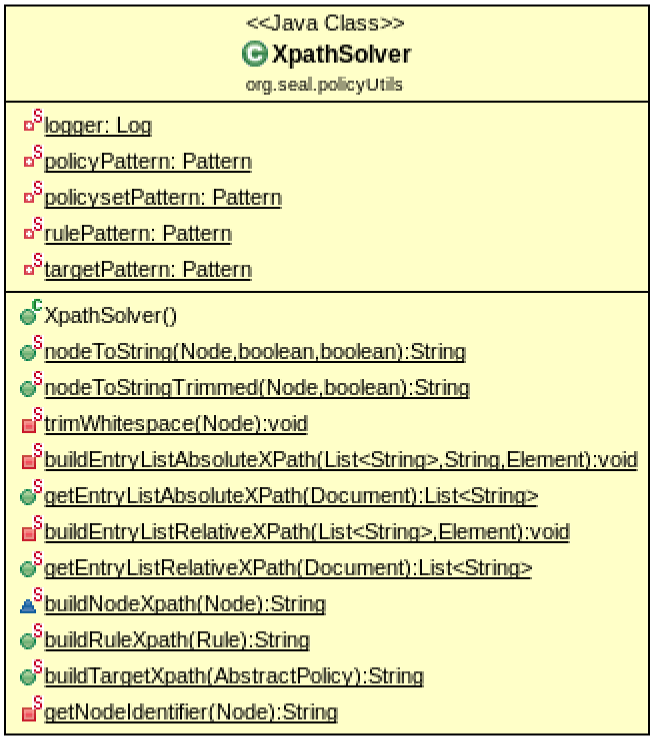
The `coverageMatrix` has a flat structure in which one policy element corresponds to a row in the `coverageMatrix`. However XACML policies have a hierarchical structure: a policy set contains a policy set target, and one or more policies or policy sets; a policy contains a policy target, and one or more rules. Therefore the structure of XACML policies must be “flattened” when mapping a policy element to a column in the `coverageMatrix`.

Simply giving each policy element an index number in the order they are visited while traversing over the XACML policy will not work. This is because the XACML engine might have implemented short-circuit evaluation (the balana implementation does have), so some policy element might be skipped when evaluating a request. For example, if an XACML policy is a policy set that consists of two policies: Policy1 and Policy2. There are two rules in Policy1: Rule1 and Rule2. And the rule combining algorithm of Policy1 is first-applicable. Suppose for a given request, Rule1 is fired(both target and condition are evaluated to be true), then the XACML engine will skip Rule2, and go on evaluating Policy2. In such scenario, the coverage information of the following policy elements will be written to the wrong columns in the coverageMatrix.

This is solved by mapping each policy element's XPath to its column index. XPath is a part of the XSLT standard. It uses path expressions to select a node an XML document. For example, the first rule in the sample XACML policy in Figure 3.1 can be selected by the absolute XPath "`/*[local-name()='PolicySet' and @PolicySetId='KMarketBluePolicy']/*[local-name()='Rule' and @RuleId='total-amount']`". The XPath expression means that starting from the root node, first look for a node whose local-name is "PolicySet", and has an attribute "PolicySetId" that equals to "'KMarketBluePolicy"; then from the child nodes of this node, look for a node whose local-name is "Rule", and has an attribute "RuleId" that equals to "total-amount". The "*" in the XPath expression means ignoring namespace prefix. The first rule can also be selected by the relative XPath "`//*[local-name()='Rule' and @RuleId='total-amount']`". The double slash at the beginning of the path expression means that this is a relative path so the node is not necessarily a child of the root node.

According to the specifications of XACML, the policy element ID attribute must be unique in an XACML policy. So each policy element can be uniquely identified by its local-name and policy element ID attribute, and only one node will be selected even if using relative XPath. And since there is no `getParent()` method in PolicySet, Policy, Rule classes in the Balana implementation of XACML engine, it is impossible to construct an absolute XPath from a policy element. Therefore, the relative XPath is used for mapping policy elements to column indices.

The module `XpathSolver` in the component `PolicyUtils` is used for getting the XPath of policy elements in an XACML policy. Figure 5.2 shows the UML class diagram of `XpathSolver`. The method `getEntryListRelativeXPath(Document)` returns XPath strings of all policy elements in the order they are visited in a recursive traversal of the XACML document. `PolicyCoverageFactory.init()` uses this method to set the `mapping`, thus getting a mapping from XPath of each policy element to their column index in the `coverageMatrix`. Latter while the tests are running, each time a policy element is touched by a test, the column index of the policy element is looked up in the `mapping`, and a `Coverage` object is created and inserted at the column index in the last row.



<<Java Class>>
XpathSolver
org.seal.policyUtils
logger: Log
policyPattern: Pattern
policysetPattern: Pattern
rulePattern: Pattern
targetPattern: Pattern
XpathSolver()
nodeToString(Node,boolean,boolean):String
nodeToStringTrimmed(Node,boolean):String
trimWhitespace(Node):void
buildEntryListAbsoluteXPath(List<String>,String,Element):void
getEntryListAbsoluteXPath(Document):List<String>
buildEntryListRelativeXPath(List<String>,Element):void
getEntryListRelativeXPath(Document):List<String>
buildNodeXpath(Node):String
buildRuleXpath(Rule):String
buildTargetXpath(AbstractPolicy):String
getNodeIdentifier(Node):String

Figure 5.2 UML Class Diagram of XpathSolver

PolicyTracer

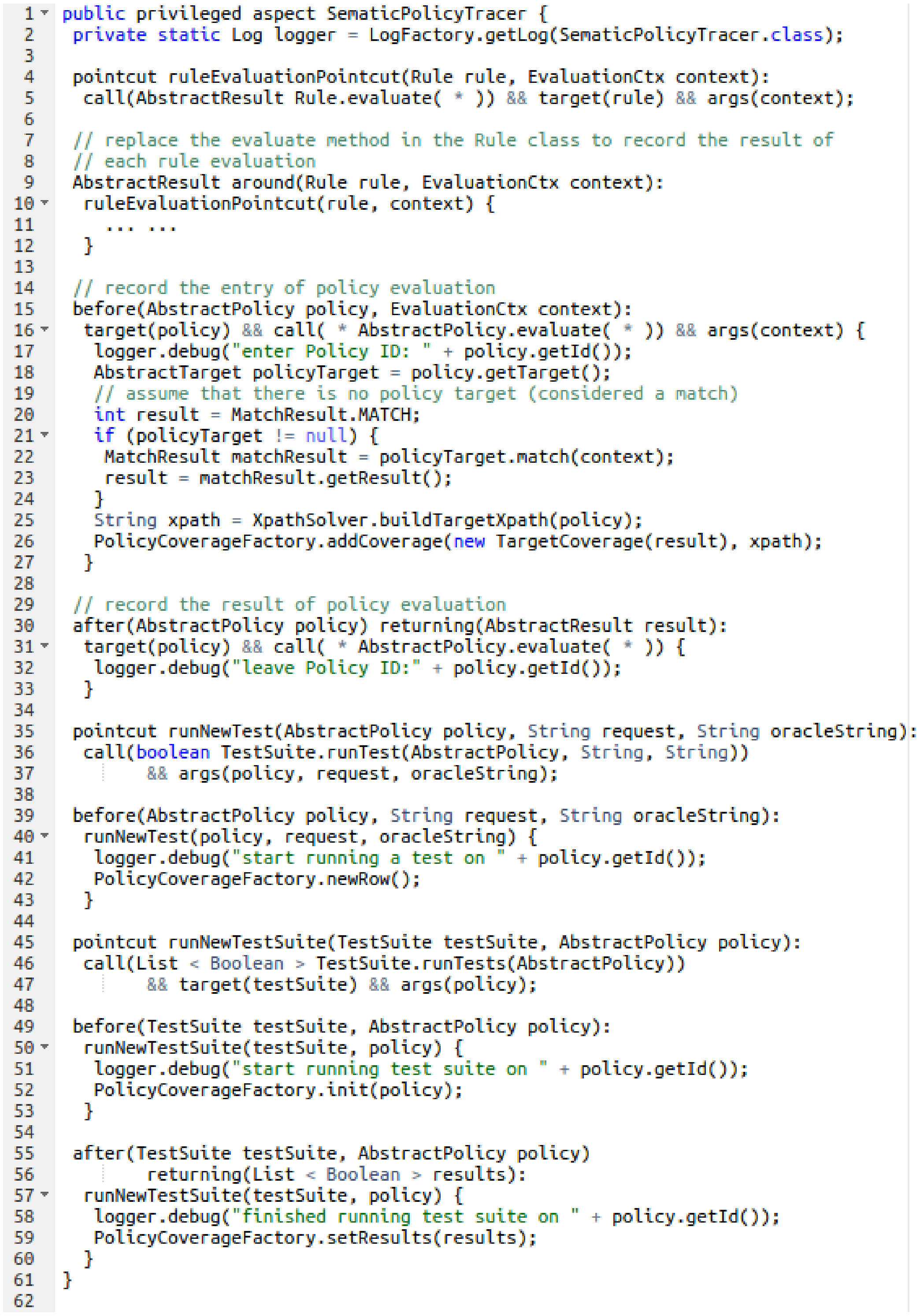
`PolicyTracer` is used to change the behavior of some methods related to the evaluation of XACML requests in the XACML engine for the purpose of collecting coverage information. This functionality is accomplished with AspectJ. AspectJ is an Aspect-Oriented Programming (AOP) extension for Java. The aim of AOP is to increase modularity by putting code that is not central to business logic and appears multiple places into one module, e.g. logging.

Basic concepts in AspectJ include join points, pointcuts, and advices. A join point is a point in a program where additional code can be joined into, e.g. method execution, object initialization, field read and write. A pointcut is an expression that matches one or more joint points. For example, in the PolicyTracer code shown in Figure 5.3, line 45-47 defines a pointcut called `runNewTestSuite`, which matches the `runtTests()` method in the TestSuite class. An advice is a piece of code that runs before, after or around a join point that matches a pointcut. During execution, when a joint point matches a pointcut, the AspectJ runtime automatically invokes the advice associated with the pointcut. In Figure 5.3, line 49-53 is an advice associated with the pointcut `runNewTestSuite`. Therefore, before `TestSuite.runTests()` is called, a message is written to log and `PolicyCoverage.init()` is called.

Figure 5.3 shows the source code of PolicyTracer. The advice associated with `ruleEvaluationPointCut` is omitted to save space.

Line 35-43 defines a pointcut `runNewTest` and an associated advice that calls `PolicyCoverageFactory.newRow()`, adding a new row in the `coverageMatrix` before every test starts.

Line 15-27 defines an advice that gets the XPath of a policy target or policy set target, creates a `Coverage` object, and calls `addCoverage()` to add the Coverage object into `coverageMatrix`.



```
1  public privileged aspect SematicPolicyTracer {
2   private static Log logger = LogFactory.getLog(SematicPolicyTracer.class);
3
4   pointcut ruleEvaluationPointcut(Rule rule, EvaluationCtx context):
5    call(AbstractResult Rule.evaluate( * )) && target(rule) && args(context);
6
7   // replace the evaluate method in the Rule class to record the result of
8   // each rule evaluation
9   AbstractResult around(Rule rule, EvaluationCtx context):
10   ruleEvaluationPointcut(rule, context) {
11     ... ...
12   }
13
14  // record the entry of policy evaluation
15  before(AbstractPolicy policy, EvaluationCtx context):
16   target(policy) && call( * AbstractPolicy.evaluate( * )) && args(context) {
17    logger.debug("enter Policy ID: " + policy.getId());
18    AbstractTarget policyTarget = policy.getTarget();
19    // assume that there is no policy target (considered a match)
20    int result = MatchResult.MATCH;
21    if (policyTarget != null) {
22     MatchResult matchResult = policyTarget.match(context);
23     result = matchResult.getResult();
24    }
25    String xpath = XpathSolver.buildTargetXpath(policy);
26    PolicyCoverageFactory.addCoverage(new TargetCoverage(result), xpath);
27   }
28
29  // record the result of policy evaluation
30  after(AbstractPolicy policy) returning(AbstractResult result):
31   target(policy) && call( * AbstractPolicy.evaluate( * )) {
32    logger.debug("leave Policy ID:" + policy.getId());
33   }
34
35  pointcut runNewTest(AbstractPolicy policy, String request, String oracleString):
36   call(boolean TestSuite.runTest(AbstractPolicy, String, String))
37        && args(policy, request, oracleString);
38
39  before(AbstractPolicy policy, String request, String oracleString):
40   runNewTest(policy, request, oracleString) {
41    logger.debug("start running a test on " + policy.getId());
42    PolicyCoverageFactory.newRow();
43   }
44
45  pointcut runNewTestSuite(TestSuite testSuite, AbstractPolicy policy):
46   call(List < Boolean > TestSuite.runTests(AbstractPolicy))
47        && target(testSuite) && args(policy);
48
49  before(TestSuite testSuite, AbstractPolicy policy):
50   runNewTestSuite(testSuite, policy) {
51    logger.debug("start running test suite on " + policy.getId());
52    PolicyCoverageFactory.init(policy);
53   }
54
55  after(TestSuite testSuite, AbstractPolicy policy)
56        returning(List < Boolean > results):
57   runNewTestSuite(testSuite, policy) {
58    logger.debug("finished running test suite on " + policy.getId());
59    PolicyCoverageFactory.setResults(results);
60   }
61 }
62
```

Figure 5.3 Source code of PolicyTracer

Line 4-12 defines a pointcut `ruleEvaluationPointCut` and an associated advice that gets the XPath of a rule, creates a `Coverage` object, and calls `addCoverage()` to add the Coverage object into `coverageMatrix`.

Line 55-55 defines an advice that calls `PolicyCoverageFactory.setResults()` to add the test results of the test suite to `PolicyCoverageFactory.testResults` after `TestSuite.runTests()` has finished.

Fault Localization

The component `Fault Localization` deals with fault localization. Figure 5.4 shows the UML diagram of this package.

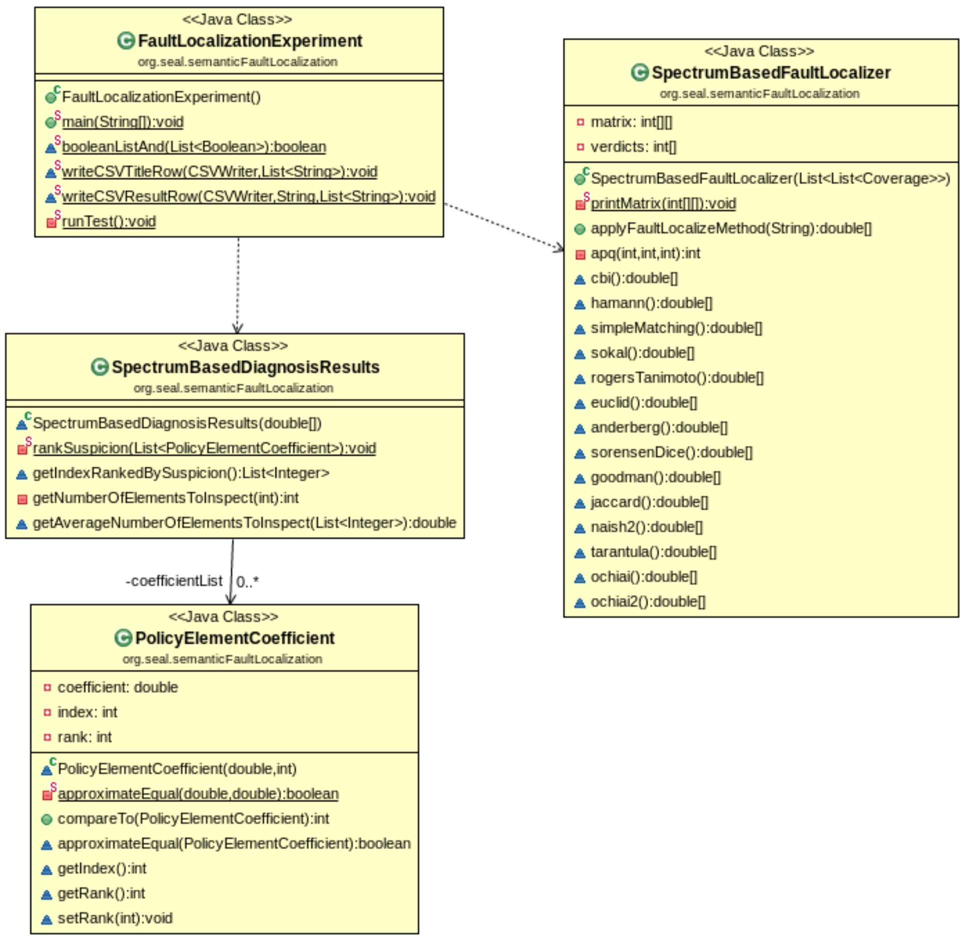

<<Java Class>>
FaultLocalizationExperiment
org.seal.semanticFaultLocalization
FaultLocalizationExperiment()
main(String[]):void
booleanListAnd(List<Boolean>):boolean
writeCSVTitleRow(CSVWriter,List<String>):void
writeCSVResultRow(CSVWriter,String,List<String>):void
runTest():void
<<Java Class>>
SpectrumBasedFaultLocalizer
org.seal.semanticFaultLocalization
matrix: int[][]
verdicts: int[]
SpectrumBasedFaultLocalizer(List<List<Coverage>>)
printMatrix(int[][]):void
applyFaultLocalizeMethod(String):double[]
apq(int,int,int):int
cbi():double[]
hamann():double[]
simpleMatching():double[]
sokal():double[]
rogersTanimoto():double[]
euclid():double[]
anderberg():double[]
sorensenDice():double[]
goodman():double[]
jaccard():double[]
naish2():double[]
tarantula():double[]
ochiai():double[]
ochiai2():double[]
<<Java Class>>
SpectrumBasedDiagnosisResults
org.seal.semanticFaultLocalization
SpectrumBasedDiagnosisResults(double[])
rankSuspicion(List<PolicyElementCoefficient>):void
getIndexRankedBySuspicion():List<Integer>
getNumberOfElementsToInspect(int):int
getAverageNumberOfElementsToInspect(List<Integer>):double
-coefficientList
0..*
<<Java Class>>
PolicyElementCoefficient
org.seal.semanticFaultLocalization
coefficient: double
index: int
rank: int
PolicyElementCoefficient(double,int)
approximateEqual(double,double):boolean
compareTo(PolicyElementCoefficient):int
approximateEqual(PolicyElementCoefficient):boolean
getIndex():int
getRank():int
setRank(int):void

Figure 5.4 UML Class Diagram of Component Fault Localization

The module `SpectrumBasedFaultLoclizer` calculates a suspicion score for each policy element basing on the coverage matrix and test results. The coverage information defined in the `Coverage` are Enum types. In order to calculate a score, the values in the Enum types must be mapped to numbers. There can be different ways for the mapping. In our implementation the “firing” criteria is used: a policy target or a policy set target is fired if it is evaluated to be true; a rule is fired only if both the rule target and condition are evaluated to be true. In `TargetMatchResult`, `MATCH` is mapped to 1, other values are mapped to 0; in `RuleDecisionCoverage`,`EFFECT` is mapped to 1, other values are mapped to 0. In addition, test results also need to be mapped to numbers. If a test passes, the test result is mapped to 1, otherwise mapped to 0.

The mapping is done in the constructor of `SpectrumBasedFaultLocalizer`. The `coverageMatrix` in `PolicyCoverageFactory` is mapped to `SpectrumBasedFaultLocalizer.matrix`, and `testResults` in `PolicyCoverageFactory`. is mapped to `SpectrumBasedFaultLocalizer.verdicts`.

Fourteen scoring methods are implemented. For example `jaccard()` implements the scoring method jaccard. In the empirical study of fault localization, sometimes it is desirable to be able to loop over a list of scoring methods. As in Java language methods are not first class functions, this is implemented by way of reflection. The method `applyFaultLocalizeMethod(String)` takes a scoring method name as input, and invoke the scoring method using reflection.

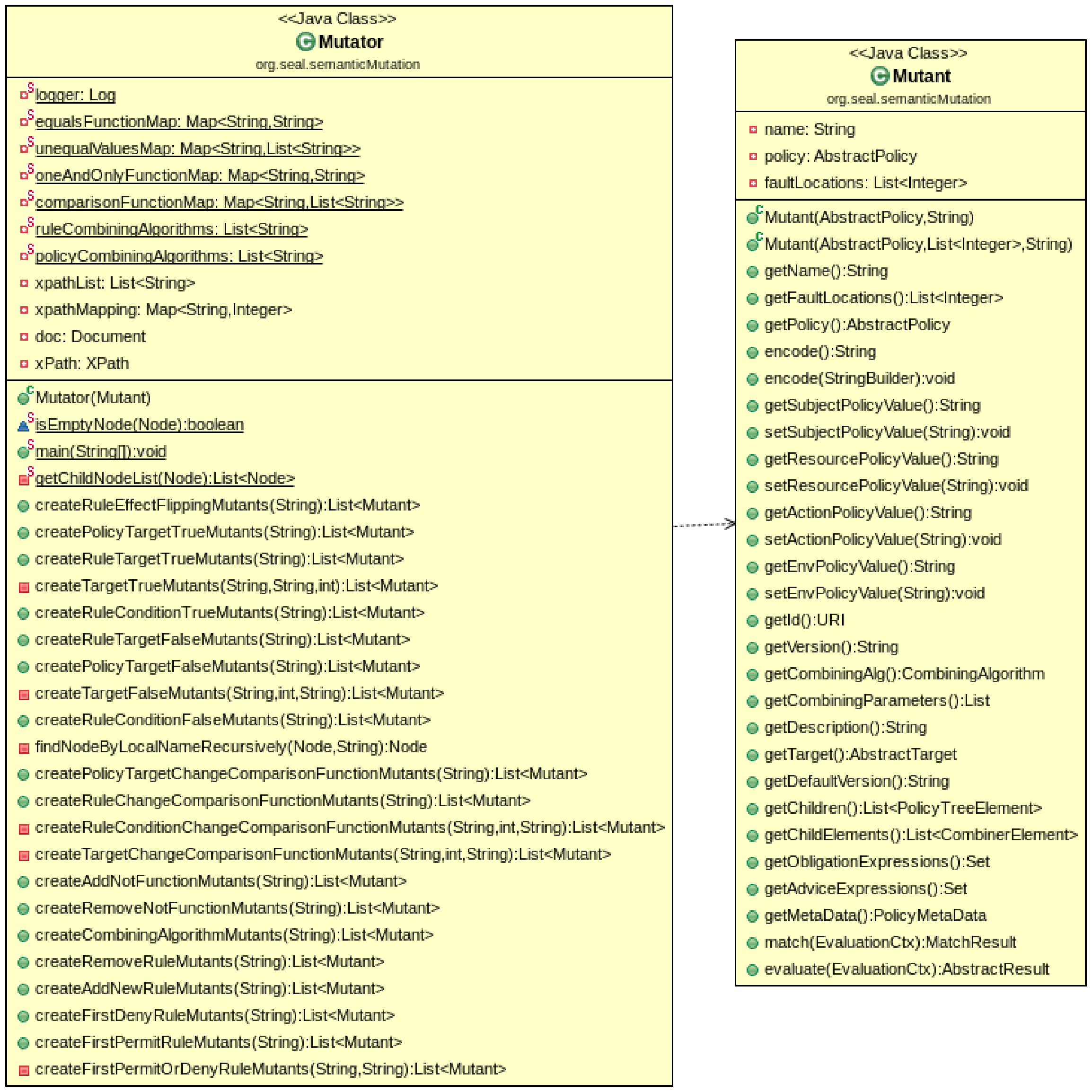
`PolicyElementCoefficient` bundles the index, suspicion score and rank of a policy element together, and used by `SpectrumBasedDiagnosisResults`, which evaluates the effectiveness of a scoring method. The constructor of `SpectrumBasedDiagnosisResults` sorts the policy elements by their suspicion score and calls `rankSuspicion(List<PolicyElementCoefficient>)` to give each policy element a rank according to their suspicion score. For example, suppose the suspicion scores of 4 policy elements are [0.5, 0.4, 0.4, 0.1], then the rank of them is [1, 2, 2, 4]. The method `getNumberOfElementsToInspect(int)` evaluates a scoring method by calculating how many policy element must be inspected in the worst case before the faulty policy element is found out. The input of this method is the index of the faulty policy element. For example, suppose in the previous example the second policy element is faulty, in the worst case three policy element must be inspected before it is found out. And the method `getAverageNumberOfElementsToInspect(List<Integer>)` calculates when there are multiple faulty policy elements, on average how many policy elements must be inspected in the worst case. Similarly, the input of this method is a list of indices of the faulty policy elements.

The `FaultLocalizationExperiment` is for empirical study of the performance of different scoring methods. It first generates or loads from hard drive a list of mutants, then perform fault localization on each mutant, using several scoring methods, and writes the number of policy elements to inspect of each pair of mutant and scoring method to a CSV file. And in the last row of the CSV file, it appends the average number of policy elements to inspect of each scoring method.

Mutation

The component `Mutation` deals with mutating XACML policies. Figure 5.5 shows the UML class diagram of package mutation.

Mutant



<<Java Class>>
Mutator
org.seal.semanticMutation
logger: Log
equalsFunctionMap: Map<String,String>
unequalValuesMap: Map<String,List<String>>
oneAndOnlyFunctionMap: Map<String,String>
comparisonFunctionMap: Map<String,List<String>>
ruleCombiningAlgorithms: List<String>
policyCombiningAlgorithms: List<String>
xpathList: List<String>
xpathMapping: Map<String,Integer>
doc: Document
xPath: XPath
Mutator(Mutant)
isEmptyNode(Node):boolean
main(String[]):void
getChildNodeList(Node):List<Node>
createRuleEffectFlippingMutants(String):List<Mutant>
createPolicyTargetTrueMutants(String):List<Mutant>
createRuleTargetTrueMutants(String):List<Mutant>
createTargetTrueMutants(String,String,int):List<Mutant>
createRuleConditionTrueMutants(String):List<Mutant>
createRuleTargetFalseMutants(String):List<Mutant>
createPolicyTargetFalseMutants(String):List<Mutant>
createTargetFalseMutants(String,int,String):List<Mutant>
createRuleConditionFalseMutants(String):List<Mutant>
findNodeByLocalNameRecursively(Node,String):Node
createPolicyTargetChangeComparisonFunctionMutants(String):List<Mutant>
createRuleChangeComparisonFunctionMutants(String):List<Mutant>
createRuleConditionChangeComparisonFunctionMutants(String,int,String):List<Mutant>
createTargetChangeComparisonFunctionMutants(String,int,String):List<Mutant>
createAddNotFunctionMutants(String):List<Mutant>
createRemoveNotFunctionMutants(String):List<Mutant>
createCombiningAlgorithmMutants(String):List<Mutant>
createRemoveRuleMutants(String):List<Mutant>
createAddNewRuleMutants(String):List<Mutant>
createFirstDenyRuleMutants(String):List<Mutant>
createFirstPermitRuleMutants(String):List<Mutant>
createFirstPermitOrDenyRuleMutants(String,String):List<Mutant>
<<Java Class>>
Mutant
org.seal.semanticMutation
name: String
policy: AbstractPolicy
faultLocations: List<Integer>
Mutant(AbstractPolicy,String)
Mutant(AbstractPolicy,List<Integer>,String)
getName():String
getFaultLocations():List<Integer>
getPolicy():AbstractPolicy
encode():String
encode(StringBuilder):void
getSubjectPolicyValue():String
setSubjectPolicyValue(String):void
getResourcePolicyValue():String
setResourcePolicyValue(String):void
getActionPolicyValue():String
setActionPolicyValue(String):void
getEnvPolicyValue():String
setEnvPolicyValue(String):void
getId():URI
getVersion():String
getCombiningAlg():CombiningAlgorithm
getCombiningParameters():List
getDescription():String
getTarget():AbstractTarget
getDefaultVersion():String
getChildren():List<PolicyTreeElement>
getChildElements():List<CombinerElement>
getObligationExpressions():Set
getAdviceExpressions():Set
getMetaData():PolicyMetaData
match(EvaluationCtx):MatchResult
evaluate(EvaluationCtx):AbstractResult

Figure 5.5 UML Class Diagram of Component Mutation

`Mutant` models a policy mutant. In the Balana implementation of XACML engine, `Policy` class models a policy, and `PolicySet` class models a policy set. And they both extends the `AbstractPolicy` class. A mutant is an XACML policy too. So the Mutant class should have all the public methods and fields of `AbstractPolicy`. Besides that, a mutant has faulty policy element, so `Mutant` should have a field that stores the indices of faulty policy elements, and a getter method for the field. Meanwhile, a mutant is created by mutating either a policy or a policy set, so a `Mutant` should behave like either a `Policy` or a `PolicySet`, depending on which one it was created from. Henceforth, `Mutant` is designed to extend the `AbstractPolicy`, and to have an `AbstractPolicy` as data member, which can be an instance of `Policy` or `PolicySet`. All the public methods of the data member are "forwarded" to inherited methods. Figure 5.6 shows a simplified UML class diagram of these four classes. And `Mutant` has a `faultLocations` field which stores the indices of faulty policy elements, and a getter method for the field.

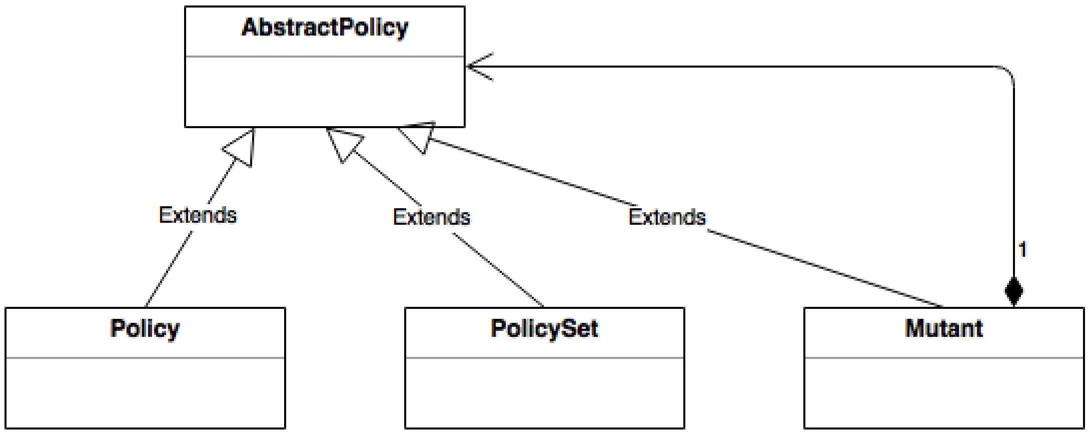


Figure 5.6 Simplified UML Class Diagram of AbstractPolicy, Policy, PolicySet and Mutant

<u>Mutator</u>

The `Mutator` module is used for creating mutants from a `Policy`, a `PolicySet` or a `Mutant`.

All the public methods starting with "create" in Mutator are methods implementing a mutation operator. For example, `createRuleEffectFlippingMutants(String)` implements the CRE mutation operator in Table 4.3, which changes the effect of a rule from Permit to Deny, or vice versa.

Because there are too many mutation operators, only a few typical mutation operators' implementation is described in this paper.

```
createCombiningAlgorithmMutants(xpathString)
    mutants = an empty list
    policyNode = xpath.evaluate(xpathString)
    originalCombiningAlgo = get rule combining algorithm of policyNode
    for each rule combining algorithm algo:
        if algo is not equal to originalCombiningAlg
            set rule combining algorithm in policyNode to be algo
            create a mutant and add to mutants
    set rule combining algorithm in policyNode to be originalCombiningAlg
    return mutants
```

Figure 5.7 Implementation of CRC

Figure 5.7 shows how the mutation operator CRC is implemented.

```
createRuleEffectFlippingMutants (xpathString)
    mutants = an empty list
    ruleNode = xpath.evaluate(xpathString)
    originalEffect = get effect of ruleNode
    if originalEffect is equal to "Permit"
        set effect of ruleNode to "Deny"
    else
        set effect of ruleNode to "Permit"
    create a mutant and add to mutants
    set effect of ruleNode to be originalEffect
    return mutants
```

Figure 5.8 Implementation of CRE

Figure 5.8 shows how the mutation operator CRE is implemented.

```
createTargetTrueMutants(xpathString)
    mutants = an empty list
    targetNode = xpath.evaluate(xpathString)
    childNodes = an empty list
    remove all child nodes of targetNode and add into childNodes
    create a mutant and add to mutants
    add all nodes in childNodes into targetNode
    return mutants
```

Figure 5.9 Implementation of PTT and RTT

Figure 5.9 shows how PTT (Policy Target True) and RTT (Rule Target True) is implemented. Note that the code makes use of the XACML specification that an empty target is always evaluated to be true.

From the above examples we can see that generally implementing a mutation operator takes three steps: find the node to change and store its state; change the node and create a mutant from the changed document; restore the node.

Repairing

The component `Repairing` deals with repairing faulty policy elements. Figure 5.10 shows the UML class diagram of this component.

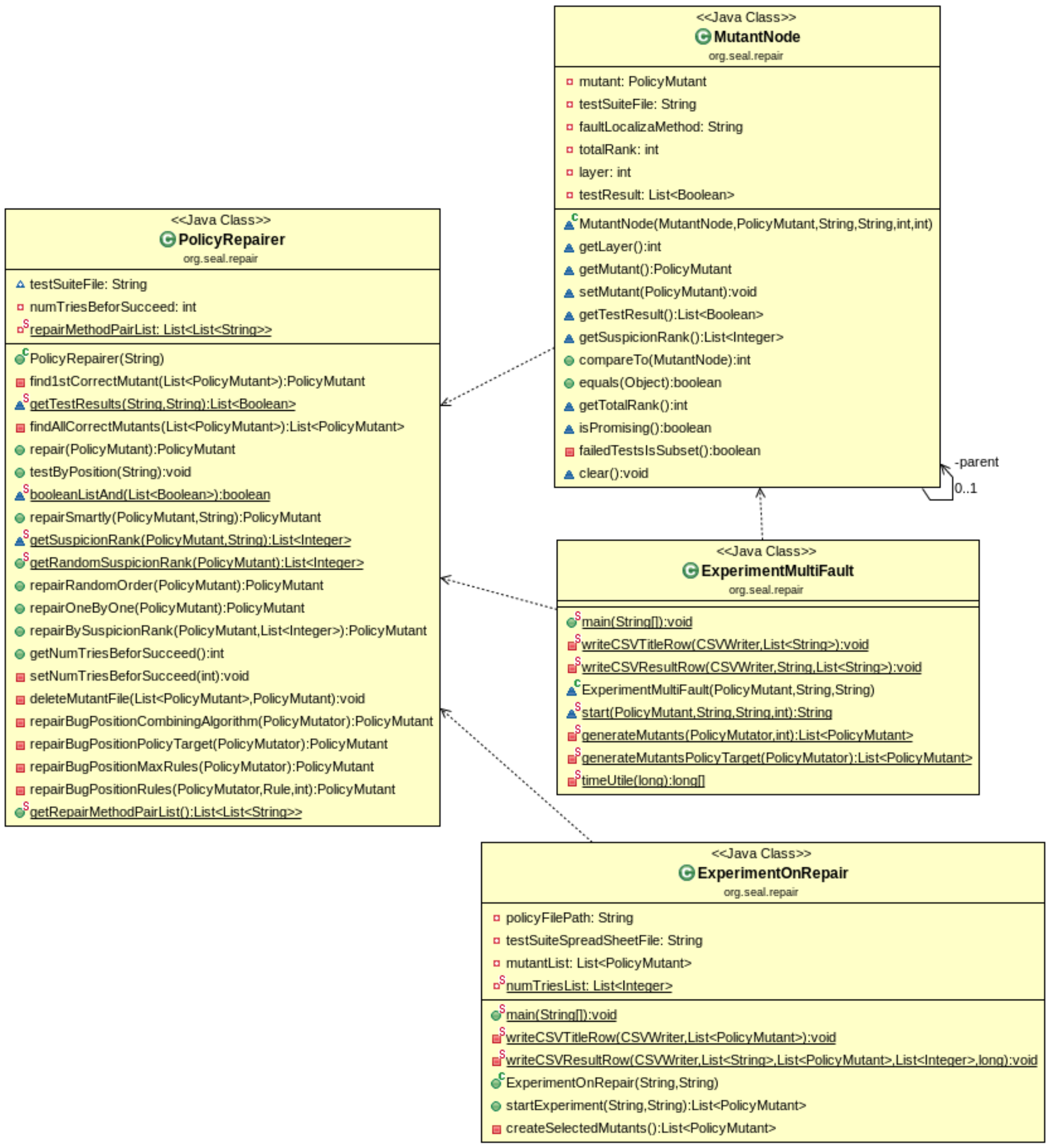
The module `PolicyRepairer` repairs a faulty policy or policy set. The method `repairSmartly(PolicyMutant, String)` repairs a faulty policy or policy set by performing fault localization, generating mutants and looking for a mutant that passes all tests. Figure 1.1 describes this process. Chapter 4 has a running example of this process.

The mutation-based repair traverses a tree where the root is the faulty policy and each node is a policy mutant. As the tree can be very large, it is pruned by excluding those branches that are unlikely leading to a successful repair. If the policy mutant in the current node passes all the tests, the repair is successful. If the set of tests failed by the policy mutant is a subset of those failed by its parent, this mutant is considered an intermediate fix. In this case, the current node shall be expanded, i.e., apply fault location and mutation to the mutant. If this mutant is not an intermediate or final fix, the node will not be expanded.

Another implementation issue is the order of mutation operators in which they are applied to the sorted policy elements. We use (PE_i) to denote the set of mutants resulted from applying mutation operators to the i-th suspicious policy element. When we run tests against the mutants in the set (PE_i) and apply mutation operators to the j-th suspicious policy element, the resultant set of mutants are denoted as (PE_i, PE_j). For $(PE_{i_1}, PE_{i_2}, \dots, PE_{i_n})$, the lesser $(i_1 + i_2 + \cdots + i_n)$is, the more suspicious this set of mutants are. This is handled by a priority queue.

The method `repairRandomOrder(PolicyMutant)` follows a similar process except that the list of suspicious policy element is not obtained from fault localization, but generated randomly. And `repairOneByOne(PolicyMutant)` is similar except that the list of suspicious policy elements is in the order they are in the XACML policy.

The module `ExperimentOnRepair` performs experiment on repairing of an XACML policy with only one faulty element. And the module `ExperimentMultiFault` performs experiment on repairing of an XACML policy with one or more faulty elements. The module `MutantNode` is used as a node in the priority queue during repairing an XACML policy with one or more faulty elements.



<<Java Class>>
MutantNode
org.seal.repair
mutant: PolicyMutant
testSuiteFile: String
faultLocalizaMethod: String
totalRank: int
layer: int
testResult: List<Boolean>
MutantNode(MutantNode,PolicyMutant,String,String,int,int)
getLayer():int
getMutant():PolicyMutant
setMutant(PolicyMutant):void
getTestResult():List<Boolean>
getSuspicionRank():List<Integer>
compareTo(MutantNode):int
equals(Object):boolean
getTotalRank():int
isPromising():boolean
failedTestsIsSubset():boolean
clear():void
-parent
0..1
<<Java Class>>
PolicyRepairer
org.seal.repair
testSuiteFile: String
numTriesBeforSucceed: int
repairMethodPairList: List<List<String>>
PolicyRepairer(String)
find1stCorrectMutant(List<PolicyMutant>):PolicyMutant
getTestResults(String,String):List<Boolean>
findAllCorrectMutants(List<PolicyMutant>):List<PolicyMutant>
repair(PolicyMutant):PolicyMutant
testByPosition(String):void
booleanListAnd(List<Boolean>):boolean
repairSmartly(PolicyMutant,String):PolicyMutant
getSuspicionRank(PolicyMutant,String):List<Integer>
getRandomSuspicionRank(PolicyMutant):List<Integer>
repairRandomOrder(PolicyMutant):PolicyMutant
repairOneByOne(PolicyMutant):PolicyMutant
repairBySuspicionRank(PolicyMutant,List<Integer>):PolicyMutant
getNumTriesBeforSucceed():int
setNumTriesBeforSucceed(int):void
deleteMutantFile(List<PolicyMutant>,PolicyMutant):void
repairBugPositionCombiningAlgorithm(PolicyMutator):PolicyMutant
repairBugPositionPolicyTarget(PolicyMutator):PolicyMutant
repairBugPositionMaxRules(PolicyMutator):PolicyMutant
repairBugPositionRules(PolicyMutator,Rule,int):PolicyMutant
getRepairMethodPairList():List<List<String>>
<<Java Class>>
ExperimentMultiFault
org.seal.repair
main(String[]):void
writeCSVTitleRow(CSVWriter,List<String>):void
writeCSVResultRow(CSVWriter,String,List<String>):void
ExperimentMultiFault(PolicyMutant,String,String)
start(PolicyMutant,String,String,int):String
generateMutants(PolicyMutator,int):List<PolicyMutant>
generateMutantsPolicyTarget(PolicyMutator):List<PolicyMutant>
timeUtile(long):long[]
<<Java Class>>
ExperimentOnRepair
org.seal.repair
policyFilePath: String
testSuiteSpreadSheetFile: String
mutantList: List<PolicyMutant>
numTriesList: List<Integer>
main(String[]):void
writeCSVTitleRow(CSVWriter,List<PolicyMutant>):void
writeCSVResultRow(CSVWriter,List<String>,List<PolicyMutant>,List<Integer>,long):void
ExperimentOnRepair(String,String)
startExperiment(String,String):List<PolicyMutant>
createSelectedMutants():List<PolicyMutant>

Figure 5.10 UML Class Diagram of Component Repairing

CHAPTER SIX: EMPIRICAL STUDIES

The empirical studies aim to answer the following research questions: (a) Can faulty XACML policies be repaired automatically? (2) How do the various scoring methods for suspicion rankings affect time performance of automatic policy repair?

In this chapter, firstly the setup of the experiments will be described, then the experiment results will be presented and analyzed.

Experiment Setup

Since faulty versions of real-world XACML policies are unavailable, the experiments rely on mutants of XACML policies. Table 6.1 shows the list of subject policies. In order to be representative, the subject policies used for experiment varies in size. The number of lines of XML code (LOC) ranges from 227 to 12,803. The number of rules ranges from 12 to 640. Continue is an access control policy for a conference management system [19]; fedora is "an open source repository system for the management and dissemination of digital content"; itrust is "a medical application that provides patients with a means to keep up with their medical history and records as well as communicate with their doctors". itrustX (X=5, 10) are expanded versions of itrust [20]. They have X times as many rules as itrust. The sizes of the policy files are believed to be a good representation of real-world applications because a very large policy is often decomposed into a number of manageable policy files.

The original subject policies are considered to be the correct version. The tests are generated automatically from the original policies by the XPA tool using the MC/DC criterion. For each test, its oracle value is the actual response to the access request produced by the original policy. The mutants of each policy are also generated automatically by the XPA tool. Each mutant is a variation of the original policy with one or two faults seeded using the mutation operators in Table 4.3.

Note that mutation analysis is a common approach to the evaluation of software testing and debugging techniques. Program mutation has the following hypotheses:

a. Mutants are based on actual fault models and are representative of real faults
b. Programs written by developers are close to being correct. This is known as the competent programmer hypothesis [21]
c. Tests that detect simple faults are also capable of detecting complex faults. This is known as the coupling effect hypothesis [22].

Empirical studies have confirmed that program mutants are indeed similar to real faults for evaluating testing techniques [23] [24]. We believe that the competent programmer hypothesis and the coupling effect hypothesis are also applicable to XACML policies. The mutants in Table 6.1 are representative of real faults because mutation operators are defined over an actual fault model of XACML policies [14].

Table 6.1 Subject policies, tests, and mutants

Subject Policy	***LOC***	***No. of rules***	***No. of tests***	***Single fault mutants***	***Two-fault mutants***
continue	229	15	27	75	5,625
fedora	227	12	31	74	5,471
itrust	1,283	64	197	324	972
itrust5	6,403	320	983	163	N/A
itrust10	12,803	640	1,965	328	N/A

The mutants are created by applying the mutation operators in Table 4.3 to the correct subject policies. Table 6.1 only includes non-equivalent mutants. An equivalent mutant has the same behavior as the original policy -- no failure would be reported when it is executed against the given test suite. Thus, the policy repair problem is not applicable to equivalent mutants. Note that the number of two-fault mutants grows quickly with the increase of policy size due to the combinations of mutation operators. The XPA tool is unable to complete the generation of all two-fault mutants for itrust5 or itrust10 because of memory and disk space constraints. Due to the large number of mutants, our experiments randomly selected 1% of the two-fault mutants of itrust, and 10% of the single fault mutants of itrust5 and itrust10.

Experiment Results and Analysis

The proposed approach is able to repair all mutants in Table 6.1, as the mutation operators in Table 4.3 are reversible – a mutant created by one operator can be mutated back to the original by the same or another mutation operator.

Table 6.2 and Table 6.3 show the average repair time of single fault mutants and two-fault mutants, respectively. “Random” refers to the scoring method that ranks all policy elements in a random fashion. It is used as a baseline method for identifying suspicious elements.

For a small policy like continue, the performance difference between different scoring methods is small, but for large policies the difference is significant. For example, for the subject policy itrust10, the best scoring method, CBI-Inc, is about 4 times faster than Tarantula, and almost 14 times faster than the random method. This is because larger policies have more policy elements so the benefits of a better ranking (a ranking in which the faulty policy element is at the top) is more obvious. Thus for a real world XACML policy, the choice of scoring method is crucial to the performance of automated repair. And by comparing Table 6.2 and Table 6.3 we can also see that repair time grows much faster with the number of faults than with the number of rules, as the number of mutants to be examined grows polynomially with the number of rules, but exponentially with the number of faults.

Table 6.2 Repair time of single fault mutants (in seconds)

	continue	*fedora*	*itrust*	*itrust5*	*itrust10*
CBI-Inc	0.053	0.073	0.496	14.813	77.165
Naish2	0.056	0.079	0.558	14.926	79.827
Sokal	0.1	0.109	0.645	23.988	124.178
Tarantula	0.051	0.068	1.367	53.517	370.067
Random	0.111	0.213	4.409	162.251	1149.35

Table 6.3 Repair time of two-fault mutants (in seconds)

	continue	*fedora*	*itrust*
CBI-Inc	0.539	0.371	6.659
Naish2	0.531	0.466	9.435
Sokal	2.240	1.042	15.265
Tarantula	0.497	0.418	46.831
Random	1.298	2.257	100.705

Figure 6.1 shows the cumulative distributions of repair time of single fault mutants for itrust and itrust5. The x-axis stands for how much time it takes at most to repair a mutant. The y-axis stands for the percentage of mutants that can be repaired within a certain time. All mutants can be repaired eventually, so all curves eventually approach 1.0. The steeper the curve is, the shorter time it takes to repair the mutants on average.

For itrust mutants (on the left), nearly 20% can be repaired instantly. Most of them can be repaired in 715 milliseconds using a non-random scoring method. CBI-Inc and Naish2 are the most efficient. About 97% mutants can be repaired in 715 milliseconds. Using Tarantula, more than 80% of mutants can be repaired in 715 milliseconds. However, it takes 8 times more time to achieve the 80% repair rate when the policy elements are ranked randomly.

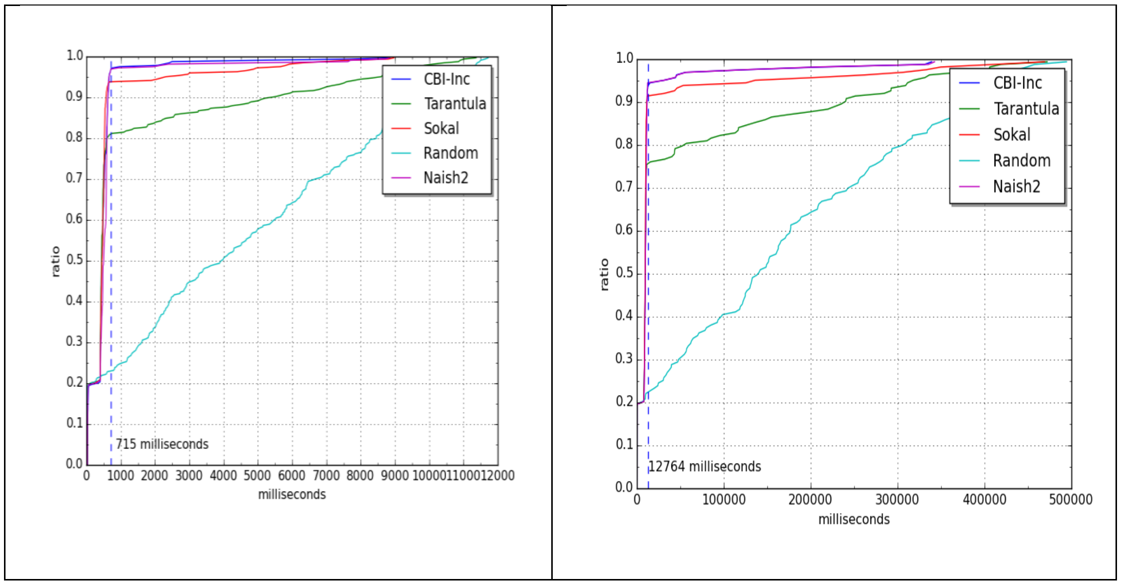


Figure 6.1 Cumulative distribution of repair time of itrust and itrust5 mutants

For itrust5 mutants (on the right), most of them can be repaired in about 13 seconds. CBI-Inc and Naish2 have almost the same performance. Within 13 seconds, both can achieve nearly 95% repair rate, while Sokal can repair more than 90% mutants and Tarantula can repair about 75%. Thus if timeout is set to be 13 seconds, the probability of fixing a faulty policy within the cutoff time is nearly 95%.

In brief, if a mutant can be fixed, a more efficient scoring method will make the faulty elements appear higher in the suspicion rankings.

CHAPTER SEVEN: CONCLUSIONS

In this paper an approach to automatic repair of XACML policies is presented. It first ranks suspicious policy elements according to the test execution information and then attempts to mutate suspicious policy elements to make all tests pass. The proposed approach also provides several scoring methods for suspicion ranking of policy elements. They are an important factor in the overall time performance of automatic policy repair, especially for policies of large size. The empirical studies show that our approach can automatically repair faulty policies with one or two injected faults and that, among the scoring methods, Naish2 and CBI-Inc have the best time performance.

This work offers the first yet promising attempt to develop techniques for automatic repair of XACML policies although the current empirical studies are inherently limited due to the general unavailability of real faults in real-world XACML policies and the use of policy mutants with only one or two seeded faults. Nevertheless, more efficient mutation-based repair techniques can be developed to deal with the search space problem for large XACML policies with a number of faults. It is worth pointing out that automatic repair is not meant to replace manual debugging of complex policies but to provide useful hints on suspicious elements and potential fixes.

In this paper, the use of MC/DC test suite of XACML policy is thought to be critical for the approach to be able to successfully repair the mutants with seeded faults. For a real-world faulty policy under debugging, however, the test suite may not be MC/DC adequate. Future research may focus on how the coverage and size of test suite affects the success rate of automatic repair. To reduce the search space of mutation-based repair, future research may also investigate coarse-grained mutation operators. The current mutation operators in the proposed approach only make a small primitive change at a time. A coarse-grained mutation can make multiple primitive changes. To do so, further research can investigate typical patterns of policy target, rule targets, rule conditions in real-world XACML policies and define coarse-grained mutation operators with respect to these patterns.

REFERENCES

[1] Hu, V. C., et al. "SP 800-162. Guide to Attribute Based Access Control (ABAC) Definitions and Considerations." Nat'l Inst. Standards and Technology, Jan (2014): 800-162.

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

[3] Martin, E. and Xie. T. Automated Test Generation for Access Control Policies via Change-Impact Analysis. In Proceedings of the Third International Workshop on Software Engineering for Secure Systems. IEEE Computer Society, pp.5-11, 2007.

[4] Martin, E. and Xie. T. A Fault Model and Mutation Testing of Access Control Policies. In Proc. of the 16th International Conference on World Wide Web. ACM, pp. 667-676, 2007.

[5] Bertolino A, Lonetti F., Marchetti E., Systematic XACML Request Generation for Testing Purposes, EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Lille 1-3 September 2010.

[6] Bertolino A., Daoudagh S., Lonetti F., Marchetti E. Automatic XACML Requests Generation for Policy Testing, The Third International Workshop on Security Testing (SecTest 2012), pp. 842 - 849. Montreal, QC, Canada, April 2012.

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

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

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

[10] Marmorstein, R., Kearns, P. Assisted Firewall Policy Repair Using Examples and History. In Proc. of USENIX Large Installation System Administration Conference (LISA), pp.1–11, 2007.

[11] Hwang, J., Xie, T., Chen, F., Liu, A.X. Fault Localization for Firewall Policies, In Proc. of IEEE International Symposium on Reliable Distributed Systems (SRDS). pp. 100–106, 2009.

[12] Chen, F., Liu, A. X. Liu, Hwang, J., Xie. T. First Step towards Automatic Correction of Firewall Policy Faults. ACM Transactions on Autonomous and Adaptive Systems, Vol. 7, No. 2, July 2012.

[13] Tang, C.M., Chan, W.K., Yu, Y.T. Extending the Theoretical Fault Localization Effectiveness Hierarchy with Empirical Results at Different Code Abstraction Levels. In Proc. of the 38th Conference on Computer Software and Applications Conference (COMPSAC'14), pp. 161-170, Västerås, July 2014.

[14] Xu, D., Wang, Z., Peng, P., Shen, N. Automated Fault Localization of XACML Policies, Proc. of the 21st ACM Symposium on Access Control Models and Technologies (SACMAT'16), Shanghai, China, June 2016.

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

[16] Xu, D., Shen, N., Wang Z. Coverage-Based Testing for Quality Assurance of XACML Policies, IEEE Transactions on Dependable and Secure Computing, Under review after revision.

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

[18] J. Jones, M. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In Proc. of the 22th International Conference on Software Engineering (ICSE'02), pp. 467–477, 2002.

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

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

[21] DeMillo, R.A., Lipton, R.J., and Sayward, F.G. Hints on Test Data Selection: Help for the Practical Programmer. IEEE Computer no. 11, pp. 34-41, 1978.

[22] Offutt, A.J. Investigations of the Software Testing Coupling Effect. ACM Transactions on Software Engineering Methodology, vol. 1, pp. 5-20, Jan. 1992.

[23] Andrews, J.H., Briand, L.C., and Labiche, Y. Is Mutation an Appropriate Tool for Testing Experiments? In Proc. 27th International Conference on Software Engineering (ICSE'05), pp. 402-411, 2005.

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

APPENDIX A

A.1 KmarketBluePolicy

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
PolicyId="KmarketBluePolicy"
RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:deny-
overrides" Version="1.0">
  <Target>
    <AnyOf>
      <AllOf>
        <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">blue</AttributeValue>
          <AttributeDesignator AttributeId="http://kmarket.com/id/role"
Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true" />
        </Match>
      </AllOf>
    </AnyOf>
  </Target>
  <Rule Effect="Deny" RuleId="total-amount">
    <Condition>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-greater-
than">
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-one-
and-only">
          <AttributeDesignator AttributeId="http://kmarket.com/id/totalAmount"
Category="http://kmarket.com/category"
DataType="http://www.w3.org/2001/XMLSchema#integer" MustBePresent="true" />
        </Apply>
        <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#integer">100</AttributeValue>
```

```
      </Apply>
    </Condition>
    <AdviceExpressions>
      <AdviceExpression AdviceId="deny-liquor-medicine-advice"
AppliesTo="Deny">
        <AttributeAssignmentExpression
AttributeId="urn:oasis:names:tc:xacml:2.0:example:attribute:text">
          <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">You are not allowed to do
more than $100 purchase
    from KMarket on-line trading system</AttributeValue>
        </AttributeAssignmentExpression>
      </AdviceExpression>
    </AdviceExpressions>
  </Rule>
  <Rule Effect="Deny" RuleId="deny-liquor-medicine">
    <Target>
      <AnyOf>
        <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">Liquor</AttributeValue>
            <AttributeDesignator
AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true" />
          </Match>
        </AllOf>
        <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">Medicine</AttributeValue>
```

```
            <AttributeDesignator
AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true" />
          </Match>
        </AllOf>
      </AnyOf>
    </Target>
    <AdviceExpressions>
      <AdviceExpression AdviceId="deny-liquor-medicine-advice"
AppliesTo="Deny">
        <AttributeAssignmentExpression
AttributeId="urn:oasis:names:tc:xacml:2.0:example:attribute:text">
          <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">You are not allowed to buy
Liquor or Medicine
    from KMarket on-line trading system</AttributeValue>
        </AttributeAssignmentExpression>
      </AdviceExpression>
    </AdviceExpressions>
  </Rule>
  <Rule Effect="Deny" RuleId="max-drink-amount">
    <Target>
      <AnyOf>
        <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">Drink</AttributeValue>
            <AttributeDesignator
AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="true" />
```

```
          </Match>
        </AllOf>
      </AnyOf>
    </Target>
    <Condition>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-greater-
than">
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-one-
and-only">
          <AttributeDesignator AttributeId="http://kmarket.com/id/amount"
Category="http://kmarket.com/category"
DataType="http://www.w3.org/2001/XMLSchema#integer" MustBePresent="true" />
        </Apply>
        <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#integer">10</AttributeValue>
      </Apply>
    </Condition>
    <AdviceExpressions>
      <AdviceExpression AdviceId="max-drink-amount-advice" AppliesTo="Deny">
        <AttributeAssignmentExpression
AttributeId="urn:oasis:names:tc:xacml:2.0:example:attribute:text">
          <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">You are not allowed to buy
more tha 10 Drinks
    from KMarket on-line trading system</AttributeValue>
        </AttributeAssignmentExpression>
      </AdviceExpression>
    </AdviceExpressions>
  </Rule>
  <Rule RuleId="permit-rule" Effect="Permit" />

</Policy>
```

A.2 A Sample Request for KmarketBluePolicy

```
<?xml version="1.0" encoding="UTF-8"?>
<Request xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
CombinedDecision="false" ReturnPolicyIdList="false">
  <Attributes Category="http://kmarket.com/category">
    <Attribute AttributeId="http://kmarket.com/id/totalAmount"
IncludeInResult="false">
      <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#integer">0</AttributeValue>
    </Attribute>
  </Attributes>
  <Attributes Category="urn:oasis:names:tc:xacml:3.0:attribute-
category:resource">
    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
IncludeInResult="false">
      <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">k</AttributeValue>
    </Attribute>
  </Attributes>
  <Attributes Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-
subject">
    <Attribute AttributeId="http://kmarket.com/id/role"
IncludeInResult="false">
      <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">blue</AttributeValue>
    </Attribute>
  </Attributes>
</Request>
```