ABSTRACT

SIVASUBRAMANIAN, DHIVYA. Automated Access Control Policy Testing Through Code Generation. (Under the direction of Ting Yu.)

Any multiuser system has to enforce access control for protecting its resources from unauthorized access or damage. One way for specifying access control is in a separate policy specification language. An access control system maintains a repository of policies, receives access requests, consults the policy and returns a response specifying whether the request was permitted or denied. However, it is challenging to specify a correct access control policy and so, it is common for the security of a system to be compromised because of the incorrect specification of these policies. There are many ways in which a policy can be checked for correctness like, formal verification, analysis and testing. In this thesis, a systematic and automatic tool for *policy testing* is provided. Testing a policy involves formulating requests that represent test cases for the policy, evaluating the policy with those test cases (requests) and comparing the responses obtained with actual expected results.

In our approach to policy testing, we generate access control *policy programs* corresponding to a policy. Dynamic analysis testing techniques are those which execute a program for different inputs to test the expected behavior. We use *concolic testing* which generates test inputs (requests) by dynamically analyzing the policy program and solving constraints to identify distinct feasible execution paths. We choose to illustrate our above technique using the most generic access control specification language, Extensible Access Control Markup Language (XACML). However, our approach to policy testing can in general be applied for testing other rule-based systems using other languages.

We conduct extensive experiments using ten policy sets to evaluate the effectiveness of our technique. We use two measures, coverage measure to check the adequacy and mutation testing for measuring the fault detection capability of the generated requests. The coverage measure shows that the request set achieves 100% structural policy coverage. We compare the fault detection capability of our request set with the existing request generation techniques. The results indicate that our request set tests different properties of the policy. This motivates the definition of a policy coverage criteria based on our method for request generation. A basic definition of the policy path coverage criteria is given. Our work is directly applicable for the quality assurance of access control policies. A coverage tool can be developed based on the defined coverage criteria. This can help in measuring the effectiveness of request sets generated by other methods also.

AUTOMATED ACCESS CONTROL POLICY TESTING THROUGH CODE GENERATION

by
DHIVYA SIVASUBRAMANIAN

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER SCIENCE

Raleigh, North Carolina

2007

APPROVED BY:

_____	_____
Dr. Peng Ning	Dr. Tao Xie

_____
Dr. Ting Yu
Chair of Advisory Committee

DEDICATION

For my parents and sister. . .

BIOGRAPHY

Dhivya Sivasubramanian was born on October 23rd,1983 in Coimbatore,India. She obtained her Bachelor of Technology degree in Information Technology from Coimbatore Institute of Technology (CIT), an autonomous institution affiliated to Anna University in 2005. After graduation she joined North Carolina State University in August 2005 for her graduate studies in the Computer Science Department.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my family, my parents and sister for their love and support for everything.

I would like to thank my adviser, Dr.Ting Yu for his guidance and support for my research work. His careful and critical comments greatly improved this thesis work. I would also like to thank Dr.Tao Xie for his invaluable advice, guidance and also for agreeing to serve on my thesis advisory committee. I also thank Dr.Peng Ning for kindly agreeing to serve on my thesis advisory committee.

I would like to acknowledge Evan Martin for his help in starting this project and also for the useful discussions.

Last, but not the least I would like to thank all my friends for their support in the course of this thesis work.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Any multiuser system has to enforce access control for protecting its resources from unauthorized access or damage. Access control is one of the fundamental mechanisms for information system security and it is widely used in operating systems, databases, networks, etc. All these systems support different applications with multiple users and every activity performed by a user or a process must be checked to see if it is authorized. An access control system determines what principals can access what resources and when.

Access control is traditionally enforced by directly hard coding into a system. However, this is tedious and becomes difficult for a large system. Also, this makes it hard to accommodate changes of security requirements in a system. Recently, access control systems increasingly separate policy from mechanisms. That is, an access control policy is explicitly specified using certain policy languages. And a system dynamically consults the policy to determine whether an access request should be granted. The advantage of this is that by separating policy from mechanism makes it easier to specify the protection requirements to be enforced on the system independent of the underlying implementation details. Also, when the security requirements on the system change later on, it is possible to easily change the policy without affecting the underlying mechanism implementing it.

The advantages of using a policy specification languages has led to the development of many specific and generic policy languages. Ponder [3] is an object oriented policy specification language for distributed systems management. Enterprise Privacy Authorization Language(EPAL) [18] is a formal language used to specify fine-grained enterprise privacy polices. Extensible Access Control Markup Language(XACML) [16] is a general purpose policy language and an access request/response language defined using Extensible Markup Language(XML) for managing access to resources. The XACML specification in its XML format enables access policies to be

transportable and also inter operable across various access control systems.

The use of separate policy specification language provides a systematic way for expressing, managing and maintaining access control policies. However, this does not by itself ensure the correctness of the specified policies. As explained below, there can be anomalies and inconsistencies in a specified policy and the security of a system is only as good as the policy.

As an example, consider the case of firewall policies. Firewalls are one mechanism for securing network resources. It is common for mis configured firewall policies to be causing problems. In examining 37 firewalls in production enterprise networks in 2004, Wool found that all the firewalls were mis configured and vulnerable [29]. In addition, the study states, "The protection that firewalls provide is only as good as the policy they are configured to implement. Analysis of real configuration data shows that corporate firewalls are often enforcing rule sets that violate well established security guidelines". The wide and continued spread of worms such as Blaster and Sapphire, demonstrated that many firewalls were mis configured, because "well-configured firewalls could have easily blocked them" [29]. There can be many anomalies and inconsistencies in the policy which make the network resources vulnerable to security attacks. Also, firewall rules are developed over a period of time. New rules are periodically added as more resources with new constraints are added to the network. It is difficult to check for conflicts or overlaps of new rules with existing rules. Similar problem exists in access control policies of enterprises and other systems. An enterprises's security policy is also revised over time as new security requirements are added. Therefore, it is critical to specify access control policies correctly, which however is a challenging problem.

There are various ways in which the quality of the policy can be assured like, formal verification, analysis and testing. Formal verification techniques can verify if a policy satisfies a particular security property [9, 31]. However, a formal representation of the policy is not scalable and properties about a policy do not exist in practice. Analysis of policies can include semantic analysis like performing a change impact analysis between two policies [6]. Testing is one practical way for checking the correctness of a policy specification. Semantic analysis techniques can be used complementary to testing.

In this thesis, we use the new approach of *policy testing* for the quality assurance of access control policies. Policy testing is the technique where the requests that a policy can receive are formulated and are evaluated against the policy and responses are got. This response is then checked to see if it is as expected. In general, once a policy is written by a security expert, they are tested by formulating a set of manually generated ad-hoc requests to check the correctness of

the policy. However, these requests are not exhaustive and all the features of the policy may not be tested. Also, it is tedious to manually formulate requests for large policies. So, there is a need for a systematic and automatic method for policy testing.

There are two existing policy testing techniques. Martin, Xie, and Yu [15] have developed a random test generation tool for XACML policies. The tests (requests) are generated as a set of all combination of attributes found in the policy. The tool represents this attribute as a bit vector and an attribute appears in the request only if the corresponding bit in the vector is set to 1. The number of requests to be generated can be user specified. To achieve adequate coverage, even in a small request set, they modify the random bit setting algorithm to ensure each bit is set at least once. This method, though simple to implement is not ensure that a policy is thoroughly tested. Martin and Xie [14] have developed the target driven test generation tool for testing XACML policies. This tool considers the policy as a hierarchical tree with the rules as the leaves. The conditions along branch in the tree leading to a rule is solved to generate the requests. This method, though outperforms the random test generation technique in terms of policy coverage and fault detection, it considers only each rule at a time and the effect of the policy as a whole is not considered.

In our approach to policy testing, we generate *policy programs* corresponding to a policy. The large amount of existing software testing techniques can be applied to this policy program for policy testing. The process of software testing is often called *dynamic analysis* because it requires that the software be executed for different inputs and the corresponding outputs be observed. This is in contrast to *static analysis* techniques like model checking which do not require the execution of the software. We use dynamic analysis techniques in our approach to policy testing. We use XACML as the policy language, but our techniques can be easily extended to other rule-based systems using different languages.

1.1 Contributions of this thesis

The following are the contributions made by this thesis,

1. The thesis proposes an automated method that uses software testing techniques for testing access control policies. The use of software testing techniques for ensuring the correctness of access control policies is novel.

2. We have developed a tool for the automated conversion of XACML policies to programs. Each of the XACML policy element is converted into a corresponding code element. This

code can be executed and software testing techniques like path coverage analysis can be used on it for policy testing. This idea of automatically generating policy programs is novel.

3. We have conducted extensive experiments to evaluate the effectiveness of our approach, and found that our method achieves 100% policy structural coverage. The mutation testing results show that our technique tests different properties of the policy than existing techniques.

4. Based on our technique, we define a stronger coverage criteria for testing access control polices. This can be used for assessing requests generated by other techniques also.

1.2 Organization of this thesis

The thesis is organized as follows, Chapter 2 provides background information on the various components of an access control system and policy specification languages. In Chapter 3, we explain our framework for access control policy testing. Chapter 4 presents the results of our evaluation and defines the new policy coverage criteria. Chapter 5 describes related work in the area access control and policy verification. Chapter 6 concludes and gives direction for future work.

Chapter 2

Access Control Policies and Enforcement

2.1 BackGround

In this chapter, we give an overview of the various components that make up an access control system and an introduction to policy specification languages. The figure 2.1 shows the various functional components of any system protecting its resources by enforcing access control. The user makes a request to the entity protecting the resources in the system, the Policy Enforcement Point(PEP). The PEP forms the appropriate access control request in a format applicable to the Policy based on the attributes of the requester, the action sought, the resource requested and the environment and gives it to the Policy Decision Point(PDP). The PDP looks up the policy that applies to the request and returns a response to the PEP. The PEP then returns the corresponding decision to the requester. The advantage of using this abstract model is that any application can use this system.

There can be various vulnerabilities in a system implementing access control. For example, the user has to first be properly authenticated into the system. Then the PEP should correctly perform the translation from the user/application specific request to that specific to the policy. This is a vulnerability because the policy specification language may be more expressive for specifying an application's security requirements. For example, XACML allows a set of subjects to request access to a set of resources. But an application can have a strict requirement that only one subject can access one resource at a time. In this cases, the PEP implementation should be correctly implemented to be aware of this restriction when performing the translation from the user's request to a policy specific request. Next, the access control policies have to correctly specify the intended behavior of the system. Also, the PDP has to perform the evaluation correctly. Among these vulner-

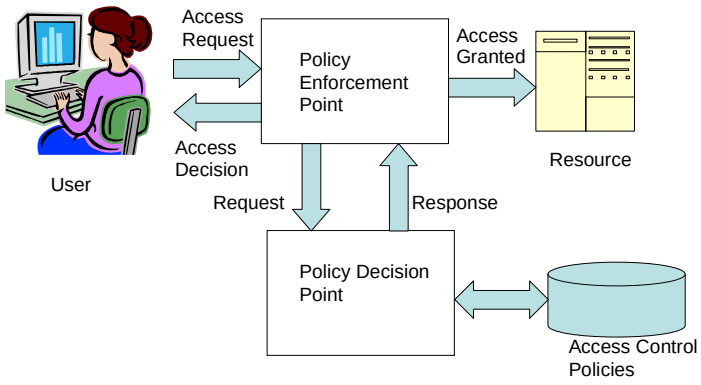

Figure 2.1: Functional Components of an Access Control System

abilities, one of the most basic requirement is to ensure that the security policy is specified correctly.

In this thesis, we focus on the problem of ensuring that the access control polices are specified correctly. A policy is considered to be correctly specified when it satisfies all the properties of the system. An example of a property is that a particular subject should not access certain resources. These properties can be explicitly and formally expressed and formal analysis techniques like resolution theorem proving can be used to prove if a property holds in a policy. However, such properties of a policy do not exist in practice and it is difficult to infer such properties in a large system. Also, the formal analysis techniques are not scalable. A practical way for ensuring the correctness of the policy is to test the policy against a set of requests and check if the responses obtained are as expected. This is the policy testing approach which is followed in this work.

2.2 Security Policy

The security policy in an access control system provides a systematic way for specifying the strategy and practices for ensuring the security, integrity and availability of resources in an information system. In this section, we will give a brief overview of policy specification languages and describe XACML which we will be using for illustrating our approach to testing.

2.2.1 Policy Specification Languages

Previously access control policies were written by hard coding directly into the program by the programmer. Later on, as the policies became more and more complicated, separate policy specification languages were developed. There are many policy specification languages and they

can be either generic or specific to applications. Generic policy specification languages are designed for enforcing access control in broad domains like distributed policy management [3], protecting the privacy of enterprises [18] , etc. Jajodia et al [10] propose a logical language for a model that allows the specification of different access control policies. They have also proposed a Flexible Authorization Manager (FAM) [11] that can enforce multiple access control policies within a single system. Besides generic policy languages, researchers have also designed models and languages for specific applications like, a model for information access control in a clinical information system [1], an access control model for work flow management [2], security policies for distributed system services [19], an access control language for web services [25],etc.

Our technique for policy testing is general and can be used for testing access control policies specified in other rule-based systems. In this thesis we present our approach in the context of one of the generic specification language XACML.

2.2.2 XACML

XACML provides a standardized way of expressing authorization polices and a standard format for expressing queries over these policies.

We have chosen to illustrate our approach to policy testing using XACML because it a general purpose specification language with various advantages for enforcing access control like,

- It is an open source standard ratified by the Organization for the Advancement of Structured Information Standards (OASIS). Because it is a standard, the various features of XACML has been examined by experts and so the specification is stable. Also, it is expected to be used widely in the industry because of its ratification.

- XACML is specified in XML format which is used for e-business applications for Electronic Data Interchange (EDI) in business-to-business and business-to-consumer transactions. Because of this, these applications can be easily configured to exchange or share XACML policies for enforcing access control.

- The specification is flexible and extensible. It is flexible since it is a generic standard and can be applied for specifying policies in all applications. It is extensible because the data types, functions, attribute types, and the way for combining multiple applicable rules/ polices can be extended. Also currently there is work on developing an XACML profile for Web Services, SAML and LDAP. This shows the language is adaptable to different environments.

- It is a portable standard. Since the specifications is in XML format, it can be used across applications.
- Conceptually it follows the PDP and PEP model which makes it applicable to many application environments.
- It supports distributed policies. The security policy of an enterprise may be enforced at different points and there is a need for specifying distributed policies.

XACML follows the abstract model as shown in figure 2.1 for policy enforcement defined by the Internet Engineering Task Force(IETF) [30] [28].

The specification defines the PEP (Policy Evaluation Point) and PDP (Policy Decision Point) as any other access control implementation. The request is given to a PEP which processes it and converts it to an XACML request format and gives it to the PDP. The PDP has access to the policies and it gets the request and determines if it has to give access to the policy or not. The PDP and PEP implementation is dependent on the application. They may be in the same application or be as separate entities on different applications or be available as a service over a network.

XACML Constructs

All XACML polices contain either a *Policy* or *Policy Set* as the basic element. A *Policy* is composed of a set of *Rules*. A set of policies or policy sets are combined to form a *Policy Set*. Figure 2.2 shows these main components and the hierarchical relation between *Policy Sets*, *Policies*, *Rules* and *Conditions* in XACML. When there are multiple rules in the policy and multiple policies / policy sets in a policy set, it is possible that a single access request can be applied to multiple rules to return conflicting access decisions. The way these conflicts must be resolved is dependent on the specific application's policy. However, XACML specifies some standard rule and policy combining algorithms for this. They are,

First Applicable : Among the set of rules (policies), this returns the effect of the rule (policy) that first evaluated to true. Here, the ordering of the rules is important.

Permit Overrides and Deny Overrides : In the set of rules (policies) in a policy (policy set), if a rule (policy) evaluates to true and if its effect is permit then the result of the rule (policy) combination is permit. If the effect of the rule (policy) is deny or if it is not applicable then all the rules (policies) in the set are evaluated to check if there is any permit rule (policy)

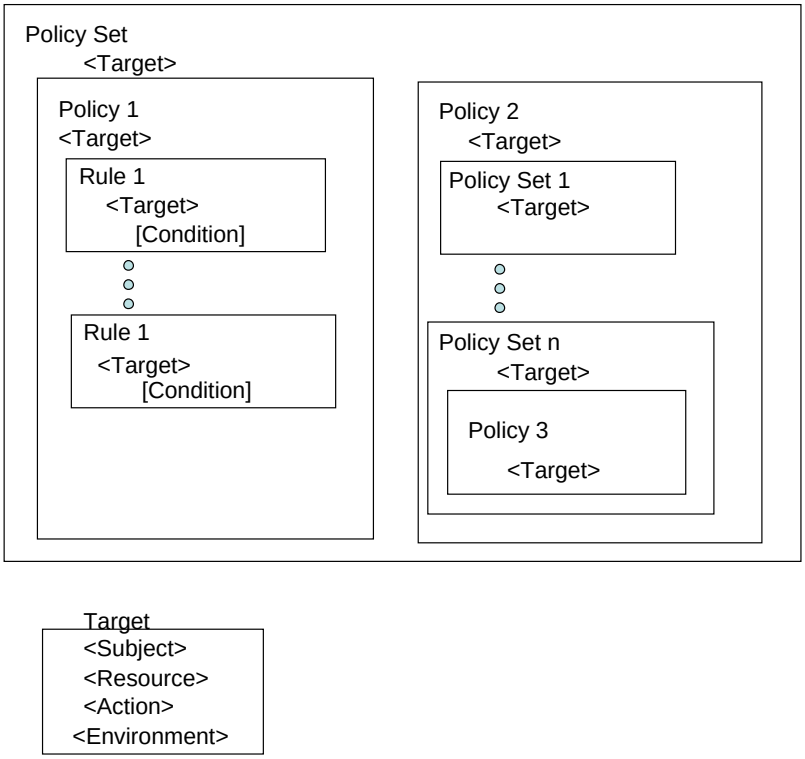

Figure 2.2: Components of XACML Policy

evaluating to true. If such a rule exists, a permit decision is returned, if not a deny decisions is returned if none of the permit rules are applicable. Hence, here permit rules are given precedence. Similarly the deny overrides rule (policy) combining algorithm is defined.

Only One Applicable : This combining algorithm is defined for combining policies in a policy set. If no policy is applicable or more than one policy is applicable, the result is defined as not applicable and in determinant. If only one policy in the policy set is applicable for a request then the result of the policy is returned.

The permit overrides and deny overrides combining algorithms can be specified to be ordered requiring that the rules be evaluated in the order in which they are specified. In addition to the above, user-defined combining algorithms can also be added.

A *Rule* is the most elementary unit of a *Policy* [16]. A rule is made of the elements, *Target*, *Effect* and the optional *Condition* element. A *Target* defines a set of *Subject*, *Resource* and *Actions* elements for which the rule is intended to apply. The *Effect* specifies the access decision as permit or deny that is returned on the successful evaluation of the rule. The *Condition* element can include complex functions that further refine the applicability of the rule. The policy itself can have a target specifying the applicability of the policy. In this case, this target can be thought of as an index into the policy. The index can be the common criteria that has to be satisfied for the set of rules in the policy to be applicable. When there are a number of policies each with a number of rules, the index of each policy helps to speed up the evaluation of a decision request by first checking the applicable policy targets and then evaluating the rules in those policies. A policy set may be used for semantically grouping policies/ policy sets. For example, grouping those policies defining authorization for a particular object/subject, etc. A policy set also includes a target element which again be used as an index for checking the applicability before evaluation of all the constituent elements.

The other essential constructs in an XACML policy are attributes, attribute values and functions. Attributes are named values of known types [16]. The Subject, Resource, Action and Environment of a given access request are described by attributes. A request will mostly contain a set of attributes. These are compared with the corresponding attribute values in the policy and an access decision is made. A request can match the attributes in the policy by using the *AttributeDesignator* [16] type which identifies attributes by their name and type. The *AttributeSelector* is used for matching a request with the attribute values in the policy through an XPath query. The attributes values can be operated on by a number of function like string comparison, date and time functions,

```
 1<Policy PolicyId="univ" RuleCombinationAlgId="permit-overrides">
 2 <Target>
 3   <Subjects> <AnySubjects/> </Subjects>
 4   <Resources><Resource> <AnyResource/> </Resource></Resources>
 5   <Actions> <AnyAction/> </Actions>
 6 </Target>
 7 <Rule RuleId="1" Effect="Permit">
 8  <Target>
 9   <Subjects><Subject> Faculty </Subject></Subjects>
10   <Resources> Grades </Resources>
11   <Actions><Action> Assign </Action>
12            <Action> View </Action></Actions>
13  </Target></Rule>
14 <Rule RuleId="2" Effect="Deny">
15  <Target>
16   <Subjects><Subject> Student </Subject></Subjects>
17   <Resources>Grades </Resources>
18   <Actions><Action> Assign </Action></Actions>
19  </Target>
20 </Rule>
21 <Rule RuleId="3" Effect="Permit">
22  <Target>
23     <Subjects><Subject> Student </Subject></Subjects>
24   <Resources> Grades </Resources>
25   <Actions><Action> View </Action></Actions>
26  </Target>
27 </Rule>
28 <!-- A final, "fall-through" rule that always Denies -->
29 <Rule RuleId="FinalRule" Effect="Deny"/>
30</policy>
```

Figure 2.3: An example XACML policy

logical functions, numeric conversions, set functions, bag functions, etc and the values returned can be compared to arrive at an access decision.

An Example Policy

In this section, we describe an example XACML policy in a university. Figure 2.3 shows a simplified form of this policy with the major XACML componenets. This policy describes the way in which the access to the grade resource is controlled in a university. The policy has an empty target, which means that by default the set of rules in the policy are applicable for any request. The target of the policy in general is used as an index for the rules. In the above example, the target could have been used to restrict the policy to be specific to a university as shown in figure 2.4.

There are three rules in the policy and a final fall through rule. The three rules are com-

```
<Target>
    <Subjects> <Subject> NC State University User </Subject> </Subjects>
    <Resources> <Resource> NC State University Academic Records
    </Resource> </Resources>
    <Actions> <AnyAction/> </Actions>
</Target>
```

Figure 2.4: XACML policy target for a university's policy

```
<Request>
    <Subjects> Student </Subjects>
    <Resources><Resource>Grades</Resource></Resources>
    <Actions> View </Actions>
</Request>
```

Figure 2.5: XACML Request

bined based on the permit overrides rule combining algorithm. This means that set of rules are combined giving precedence to rules with an effect of permit.

XACML Request and Response context

The XACML Request and Response context specifies the standard format with which a request and response from the system is got. The figures 2.5 and 2.6 shows a simplified form of an XACML Request and Response.

The request enables a set of subjects, resource and action elements to be specified. In this example, the student requests access to view the grade resource.

The response is returned on evaluating the request against the set of rules in the policy. In this example, a decision is returned on matching the attribute values in the request with the attribute values in the rule 3 in the policy.

```
<Response>
    <Result>
    <Decision>Permit</Decision>
    </Result>
</Response>
```

Figure 2.6: XACML Response

Chapter 3

Policy Testing Framework

In this chapter, we describe the general framework followed for policy testing. We also describe our technique for policy testing and show how it fits into this framework.

Figure 3.1 shows the general framework for testing a policy specification. The input to the framework is the access control policy that is to be tested. In the request generation phase, this policy is converted to a format suitable for testing. The output is results about the quality of the request set generated. The various phases are, request generation, testing policy against these requests and evaluating the quality of request set. These components are described in detail below,

Request Generation Process: Here, a policy is taken as input and the optimal number of requests needed for testing the policy is generated. This phase corresponds to test input selection in general software testing. As in test selection, some form of heuristics must be used to restrict the test cases generated.

Test Policy against Request: In this phase, we evaluate the generated request set against the policy to collect statistics of the generated test cases, which are later used to evaluate the quality of the test suite.

Compare and Evaluate Quality of Request Set: Here two techniques are used for evaluation. We find the policy structural coverage and the mutant kill ratio of the requests generated by our approach. We compare and analyze these results with existing request generation techniques.

In this chapter, we describe the first component - our model for request generation and Chapter 4 discusses the rest of the components - testing, evaluation and analysis of the results. We use XACML policy specification language for describing our framework.

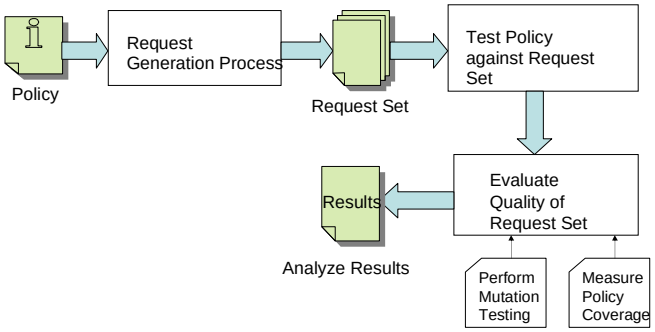

Figure 3.1: Policy Testing Framework

3.1 Policy Model

In this section, we describe how we model an XACML policy for generating requests. The overall idea of our approach is to consider the ordering and relationship between rules in the policy when generating requests. Consider an XACML policy shown in figure 2.3 for access to a grade book in a university. Here, the resource to be protected is Grades. The roles in the system are Faculty and Student. The various actions that the roles can perform on the resource are View and Assign.

The example 2.3 shows a single policy with the following rules,

1. Faculty can Assign and View Grades
2. Students cannot Assign Grades
3. Students can View Grades
4. All other requests will be denied

An XACML request to access the Grade resource will contain the *subject/s* that is the role/s making the request, the *resource/s* being requested for and the *action/s* requested on it.

The problem of policy testing is to find all possible requests that a policy can receive and check if the response is as expected. A request is a subset of the set of attributes representing the subject, resource, action and environment elements of the policy. So, all possible combinations of the sum of attribute sets will give all possible requests that a policy can receive. In general the number of possible requests that a policy can receive is 2^n where n is the number of attributes. In other words, the request to a policy is a bit vector of length n and 2^n different requests can be generated. For the above simple example with 5 attributes, the exhaustive set of requests to be

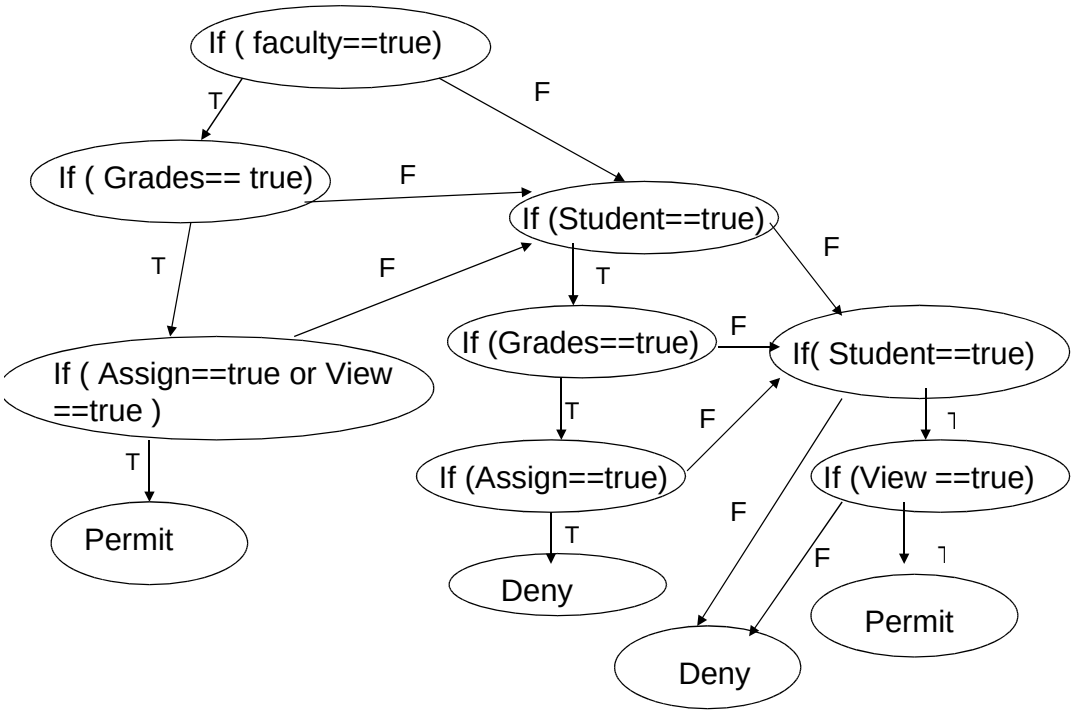

Figure 3.2: Control Flow Graph

tested is $2^5 = 32$. This technique though exhaustive, is inefficient and the number of test cases grows exponentially with the increase in number of attributes. Also, in XACML, the attributes correspond to roles, resources and actions sought, and the number of attributes directly map to the size of the application. According to a case study of the role based access control system of a bank [22] , the number of roles in an organization is 2-3% of the user population. So, generating test cases based on all combinations of attributes in a policy is not scalable.

Alternatively, in our approach, we consider the structure and semantics of the policy for generating and restricting the requests. Specific details of this approach is given in the next section. Here, we introduce the terms and concepts we use. Listing 3.1 shows a way in which the above example policy can be represented as a sequence of conditional statements. This shows the order in which the rules are considered when evaluating a request against the policy. This can be thought of as a program with a sequence of instructions. Thus, this corresponds to a *Policy Program*. White-box testing is a technique in which the internal structure of a program is examined for testing it. Now, given a policy program, white-box testing techniques can be used on the program. A common method of white-box testing is to examine the control flow graph of a program. A *control-flow-*

graph of a program is a directed graph showing all the execution paths in the program. The nodes represent the blocks of code without any branches and the edges represent jumps in the control flow. Figure 3.2 shows the control flow graph of the program corresponding to the example policy. Control flow graph of a program is used in many static analysis tools and compiler optimizations.

Listing 3.1: Faculty Policy code

```
1  if(role == Faculty)
2          if(resource == Grades)
3                  if(action == Assign or View)
4                                  return Permit
5
6  if(role == Student)
7          if(resource == Grades)
8                  if(action == Assign)
9                                  return Deny
10
11 if(role == Student)
12   if(resource == Grades)
13                 if(action == View)
14                         return Permit
15
16 return Deny //default return value
```

A coverage metric specifies when to stop a testing process. For testing a program, one structural coverage metric that is commonly used is *path coverage*. In path coverage, the criteria to stop testing is when each path in the program is traversed at least once. The problem with this approach is that the number of paths even for a simple program can be large. For example, if the input to a program is an integer, then the number of paths that the program can take is infinite. In our case, the program is a series of conditional statements. The output of each conditional statement is a true or a false outcome. So, for the above example, there are 9 conditions in the program and the number of paths is 2^9 = 512. In general, for a policy, if there are n rules (with subject,resource and action elements) in a policy program, the number of paths is 2^{n*3}. We see that the number of paths in the program grows exponentially with the addition of a conditional statement. Path coverage, as such is not used as the only metric for measuring the adequacy of testing. Usually, some heuristics are used to restrict the number of paths. Our approach to reduce the number of paths and generate

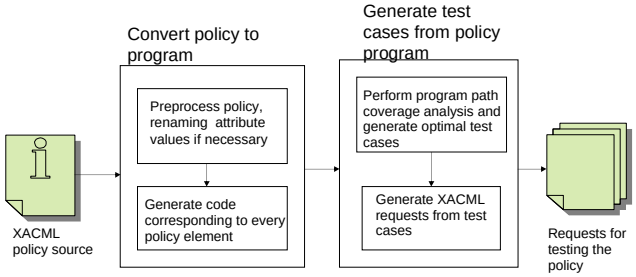

Figure 3.3: Request Generation Process

test cases is to use concolic testing (combination of concrete and symbolic executions) [23] to detect distinct paths as explained in the subsequent sections. This is done by solving the constraints which correspond to conditions represented by each rule.

3.2 Request Generation Process

The various components of the request generation process are code generation, test case generation and request formulation. This is shown in figure 3.3. The input to this step is the policy which is to be tested, the output will be a set of test cases that can be used to test the different paths in the policy. Also, an intermediate product is the policy program which can be used for other purposes like semantic code analysis as explained in chapter 6.

3.2.1 Generating XACML Policy Programs

In this section, we describe the basic idea of converting an XACML policy to code. The next sections describes the general algorithm for this. An XACML policy can be thought of as a set of predicates combined by logical operators. The predicates are the rules and the variables in the predicate correspond to attribute values in the policy. A policy is statically analyzed by parsing it and collecting all the attributes. Note that these details can be obtained from the policy writer from some specification, but we assume that such a specification is not available and we only have the policy that is to be tested. These attributes are then classified according to the type as subject, resource, action and environment. XACML represents a rule as a condition over these attribute values. A rule can be represented as a conditional statement. So, an entire policy can be represented as a series of conditional statements. All the attributes in the policy are declared as boolean variables and they can be set to true to show their presence in a rule.

As a practical example, we consider the modified form of the policy that is used by meta access management system and fedora for controlling access to objects and data streams. The XACML policy is given in the appendix in figure A.1. The policy has one policy set with four rules in the policy. The order in which the rules are to be combined is using deny overrides rule combining algorithm. The four rules are,

The target of the policy is the condition checking if the access requested is for this mod-fedora object.

1. Deny any access if client ip address is not 127.0.0.1. This policy essentially checks if the environment attribute client ip address is 127.0.0.1. If yes it grants access and proceeds to evaluate the next rule. If no it returns a deny decision.

2. Deny any access to objects or data streams, which are either inactive or deleted, unless subject has administrator role This rule first checks if the requested resource is inactive or deleted. If true then it checks if the subject accessing the resource is the administrator and it gives access.

3. Permit access to inactive data streams if the user role is special user.

4. Deny access to POLICY data stream unless subject has administrator role. Similarly, here also only if the subject is an administrator, it is given access to the resource.

The generated code corresponding to the above policy is,

Listing 3.2: Policy code for example

```
1  public class mod_fedora{
2          public static String EvaluatePolicy(boolean[] pRequestArray){
3          boolean _127_0_0_1,administrator,special_user,Inactive_object,
4          Deleted_object, Inactive_datastream,Deleted_datastream,
5          urn_fedora_names_fedora;
6
7          _127_0_0_1=pRequestArray[0];
8          administrator=pRequestArray[1];
9          special_user = pRequestArray[2];
10         policy = pRequestArray[3];
11         Inactive_object=pRequestArray[4];
12         Deleted_object=pRequestArray[5];
```

```
13          Inactive_datastream=pRequestArray[6];
14          Deleted_datastream=pRequestArray[7];
15          urn_fedora_names_fedora=pRequestArray[8];
16
17      if(((urn_fedora_names_fedora==true))){   //Policy Target
18                              Assert(false);
19                  if(!(_127_0_0_1==true )){     //rule 1
20                    Assert(false);
21
22                  }
23  }
24
25                  if(((Inactive_object==true)){
26                  Assert(false);
27                  if(!(administrator==true )){   //rule 2
28                    Assert(false);
29                  }
30          }
31          if((Deleted_object==true)){
32          Assert(false);
33                  if(!(administrator==true )){   //rule 2
34                    Assert(false);
35                  }
36          }
37          if((Inactive_datastream==true)){
38            Assert(false);
39                  if(!(administrator==true )){   //rule 2
40                    Assert(false);
41                  }
42          }
43          if((Deleted_datastream==true)){
44               Assert(false);
45                  if(!(administrator==true )){   //rule 2
46                    Assert(false);
47                  }
48          }
49
```

```
50                  if(policy==true){   //rule 4 - reordered
51                    Assert(false);
52                          if(!(administrator==true)){
53                              Assert(false);
54
55                          }
56                  }
57
58              if(special_user==true){ //rule 3 - reordered
59                Assert(false);
60                      if(Inactive_datastream==true){
61                        Assert(false);
62
63                      }
64              }
65              return "Not_Applicable";  //default rule
66  }
```

The program shows that the attributes in the policy are converted to boolean variables in the program. The input to the program is the boolean array *pRequestArray*. The presence and absence of the attributes in the policy can be controlled by setting the index in this input array.

Assertions are a software engineering technique that can be used to systematically prove the correctness of a program [7]. Java provides this feature by means of the assert statement which takes a boolean expression and can assert true or false. The jCute testing tool, looks for these assertion violations in a program. So, to generate test cases for testing the *policy program*, we add assert statements after every rule or to have more specific inputs, add assert statements after every condition check in the rule. This can be seen in example listing 3.2. Adding these assertions drives the program execution through these paths. As an optimization to get more optimal requests from jCute, the conditional statements in the program with OR's can be split up into separate if statements. This is shown in the example where the rule 2 is split into four conditional statements.

Converting permit overrides and deny overrides combining algorithms

In our approach, the permit overrides and deny overrides algorithms are converted to their first applicable forms. The permit overrides algorithm can be converted to first applicable by reordering the rules to have those with the effect of permit first and then those with the rules with

an effect of deny. Similarly, the deny overrides algorithm can be converted to its first applicable format.

The reason for doing this is that, when exploring the various paths in the program, the use of the permit overrides and deny overrides algorithms may miss certain paths, while this re-ordering makes it simple to cover these paths in the program. In the above example in listing 3.2, the rule with an effect of permit is re-ordered to be after all the rules with an effect of deny.

Algorithm for translating an XACML policy to a program

In this section we describe an algorithm for converting a policy to program. Given an XACML policy, the following describes the step by step procedure for converting it to program.

Step 1: Pre-process attribute values: Do a static analysis of the policy - parse the policy, collect attributes, remove duplicate attribute values and process the attribute values making them suitable for use as a variable name in a program.

Step 2: Maintain mapping between code and policy - write the attribute values to a file in a format that will preserve the meaning of the attributes like subjects, resource and action.

Step 3: Generate code corresponding to each policy element. The policy set target, policy target, rule target and rule conditions are read in order and converted to corresponding condition equivalents preserving the ordering between the rules.

Step 4:Add the default cases, and the test driver code needed for generating the various test cases.

The algorithm is as follows,

Listing 3.3: Process each Policy Element

```
1 processPolicyElement(Node  aNode){
2         if(aNode == PolicyTarget Element)
3                 PolicyORPolicySetTarget(aNode)
4         if(aNode == Policy Element)
5                 OnePolicy(aNode)
6         if(aNode == PolicySet Element){
7                 for every child node of PolicySet
8                         if( childNodeOfPolicySet==PolicySetTarget)
9                           PolicyORPolicySetTarget(childNodeOfPolicySet)
```

```
10                          if(childNodeOfPolicySet == Policy)
11                                  OnePolicy(childNodeOfPolicySet)
12                          if(childNodeOfPolicySet==PolicySetElement)
13                            processPolicyElement (childNodeOfPolicySet)
14          }
15  }
```

Listing 3.4: Process PolicyORPolicySetTarget Element

```
1  PolicyORPolicySetTarget(Node aNode){
2
3    subject= aNode.getSubjectElementOfTarget()
4          printCodeCorrespondingToSubject( subject)
5
6          resource= aNode.getResourceElementOfTarget()
7          printCodeCorrespondingToResource(resource)
8
9          action = aNode.getActionElementOfTarget()
10         printCodeCorrespondingToAction(action)
11 }
```

Listing 3.5: Process Policy Element

```
1  OnePolicy(Node aNode){
2          Target = aNode.getTargetOfPolicy()
3          PolicyORPolicySetTarget(Target)
4
5    if(aNode.PolicyRuleCombiningAlgorithm == PermitOverides)
6      PermitOveridesCombiningAlgorithm(aNode)
7
8          if(aNode.PolicyRuleCombiningAlgorithm == DenyOverides)
9            DenyOveridesCombiningAlgorithm(aNode)
10
11         if(aNode.PolicyRuleCombiningAlgorithm == FirstApplicable)
12                  FirstApplicableCombiningAlgorithm(aNode)
13 }
```

Listing 3.6: Permit Overides Combining Algorithm

```
PermitOveridesCombiningAlgorithm(Node aNode){
        Queue RuleQueue;
        for every rule in policy
                Rule = aNode.getRuleSet()
           if(Rule.Effect == Deny) {
                        RuleQueue.add(Rule)
                        continue
     }
                OneRule(Rule)
        for every rule in Queue
                OneRule(Rule)
}
```

Listing 3.7: One Rule

```
OneRule(Rule aRule){
        Target = aRule.getTarget()
        OneTarget(Target)
        if((Condition =Rule.getCondition()) != null)
                OneCondition(Condition)

Class aFunction{
String nameoffunction
LinkedList parametersList
}

Queue ConditionQueue
OneCondition(Condition aCondition){
aFunction oneFunction
for each child of condition
  oneFunction.name =  conditionChild.name
        if(conditionChild==anAttribueValue)
        Func.parametersList.add(conditionChild.anAttributeValue)
        if(conditionChild==aCondition){
                ConditionQueue(aCondition)
                OneCondition(aCondition)
```

```
22      }
23          for each function in ConditionQueue
24                  writeCorrespondingFunctionToFile()
25  }
```

Listing 3.3 shows the overall algorithm for processing each policy element. The function $processPolicyElement$ takes each element of the policy as input. It starts from the top most policy element. For each of the elements, the type of the element is determined as Policy Set or Policy Set Target or Policy Target or Policy and the corresponding sub-procedure is called. The procedure recurses in the case if there is a policy set inside a policy set.

Listing 3.4 shows how the target of the policy and policy set is being handled. The target is made of the subject, resource and action elements. Each of these elements correspond to a condition to be checked. So the code corresponding to this is generated.

Listing 3.5 shows how the various rules in the policy are combined using the three different combining algorithms, permit overrides, deny overrides and first applicable.

Listing 3.6 shows how the permit overrides algorithm is handled. Permit Overrides algorithm keeps evaluating the rules until a permit decision is returned. A policy with permit overrides combining algorithm can be converted to first applicable by re-ordering the rules based on their effect. The rules which have an effect of permit are put before rules which have an effect of deny so that the semantics of the ordering remain the same. In the algorithm, there is a queue to which any rule with a deny effect is encountered is added when first traversing the policy. If the effect of the rule is permit then the code corresponding to the rule is printed out. If the effect of the rule. After the end of the policy is reached, the permit rules are printed.

Listing 3.7 shows how each rule is converted to a code. Each rule is made up a target which is handled by the function in Listing 3.4. The rule can have a condition and this has to be converted into another conditional statement. A condition can have different parameters depending on the function defined. They are specified in the list.

3.2.2 Dynamic Policy Program Analysis

The purpose of converting the XACML policy to a program is to take advantage of the large amount of software testing techniques available. So, testing a policy reduces to the problem of unit testing the policy program.

Testing techniques are classified as *dynamic* and *static*. Static techniques test a software

```
if(((urnFedora==true))){
        if(!(127_0_0_1==true )){
                Assert(false);
                  return
"Deny";}} if((((inactiveObj==true) ||
(delObj==true)
||(inactiveDS==true) || (delDS==true))){
          if(!(admin==true )){
                  Assert(false);
                   return "Deny"; }}
          if(policy==true){
          if(!(admin==true))
{                 Assert(false);
                  return "Deny";  }}
           if(splUsr==true)
{         if(inactiveDS==true){
                    Assert(false);
                   return "Permit";}}
return   "NotApplicable" ;
```

127_0_0_1	admin	splUsr	inactiveObj	policy	delObj	inactiveDS	delDS	urnFedora
False	False	False	False	False	False	False	False	False
False	False	False	False	False	False	False	False	True
True	False	False	False	False	False	False	False	True
True	False	False	True	False	False	False	False	False

Figure 3.4: Request Generation from Program

without executing it but performing activities like inspection, symbolic execution and verification. Dynamic techniques test a software by generating various test inputs for executing the software and checking the results obtained. Our approach to policy testing follows the dynamic testing approach. There are many methods of dynamic program testing each depending on the specific application. In this thesis, we use concolic testing [23] for policy testing.

Concolic testing is a systematic and scalable method for program testing. The goal in concolic testing is to generate data inputs that would exercise all feasible execution paths of a sequential program [23]. The essential idea is to use concrete values as well as symbolic values as inputs for a program and execute the program both concretely and symbolically. A concrete value is a specific value for an input variable in a program. A symbolic value is a symbolic name for an input variable in a program. The execution path followed by a program on using a concrete input value is called concrete execution. In symbolic execution, an input variable is assigned a symbolic name and the program's execution path is followed. The result of this execution will be an equation in terms of input variables, which if satisfied, will lead the execution on the symbolically executed path. In the example program in listing 3.8, the input variable is x and it can be assigned the symbolic name x as $x = x$. On reaching the second statement, y has the symbolic value $y = x * x$. Now, for the if statement on line 3 to take the true branch, the inequality $x * x > 12$ must hold. Similarly, the inequality to hold in line 4 is $(x * x) + 2 > 14$.

Listing 3.8: Symbolic Execution

```
1 read(x)
```

```
2 y = x^2
3 if(y > 12)
4       Assert(false) // an error
```

The symbolic execution followed by the concolic testing algorithm is the same as the one defined above except that the algorithm takes the path that the concrete execution takes. At each branch point, during this execution, constraints over the symbolic values are collected. This is called *symbolic constraint* [23]. So, at the end of the execution, the algorithm will have a sequence of symbolic constraints corresponding to each branch point in the program. The conjunction of these constraints is called the *path constraint* [23]. It is to be noted that," all input values that satisfy a given path constraint will explore the same execution path" [23].

The concolic testing algorithm repeatedly generates inputs to traverse distinct execution paths using a depth first search strategy. The algorithm starts of with a randomly generated input. This input is used to first execute the program (concrete testing), simultaneously during this execution, the path followed by the program is modeled as symbolic constraints (symbolic execution). These constraints will be used for generating the subsequent inputs. Among the symbolic constraints collected, a constraint is picked and negated to generate the next path constraint. This way, inputs for all distinct execution paths can be found and hence redundant test cases can be avoided.

We use the tool jCUTE [24], a concolic unit testing engine for testing the policy programs. This tool logs the inputs that led to the feasible execution paths. These inputs form the optimal, non-redundant test cases for testing the program. We chose the path analysis based concolic testing for policy testing because, we believe that the test cases generated by this method could capture the type of errors more commonly done in a policy specification for the following reasons. The rules are a series of conditional statements and concolic testing easily solves each of the constraints. The addition of new rules to a policy does not change the previous rule and so new test cases can be added to the existing test cases.

The figure 3.4 shows the mod fedora policy program and the set of inputs to it generated by jCUTE by solving the constraints. The sequence in the inputs generated in the above program shows how jcute solves the constraints to generate inputs. In this program, all the input variables are boolean. Initially, the all the input variable are assigned a false value(This can also set to randomly assign true false values to the variables). For this input the program follows takes the default case, the not applicable case. During the previous execution, the constraints in each of the if statements encountered is collected. Among these, one constraint is chosen and it is negated. Here,

urnFedora==true is the negated constraint. This generates the next input to the program. This way, each of the constraints are negated to drive the execution along distinct paths.

3.2.3 Request Reduction

We do not do any request reduction because the requests returned by jCute already represent the reduced set of requests. Doing a greedy reduction as mentioned in [15] does not cause any reduction in the number of requests because each of the requests covers a single path in the program.

Chapter 4

Evaluation

In this chapter, the requests generated by our approach to request generation are evaluated. To perform the evaluation, the two measures, policy coverage and mutation testing are used. The policy coverage measures the adequacy of the requests generated and uses the existing structural policy coverage criteria. The other measure, mutation testing determines the fault detection capability of the request set and for this existing policy mutation testing tools are used. We present results of the experiments and analyse the results.

4.1 Coverage Criteria

Testing by itself can only show the presence of bugs and cannot prove that a particular property always holds. So, it is difficult to specify when a testing process is complete. It is necessary to define a test criterion before testing any software. A test criterion defines a stopping condition for the testing process. It enables identification of an adequate test. The test adequacy criteria is dependent on the nature of the application being tested. Some specific features and properties of the application may need to be tested more and so, accordingly the criteria has to be defined. Testing involves generating a set of inputs for checking the different executions of a program. Definitions of the basic components in any testing process are,

Test Case An input using which the program has to be tested. In policy testing, a request is the input to the policy that is to be tested.

Test Set The set of all test cases that are tested against the program under test. In policy testing, the request set represents the set of test cases with which the policy has to be tested to satisfy the

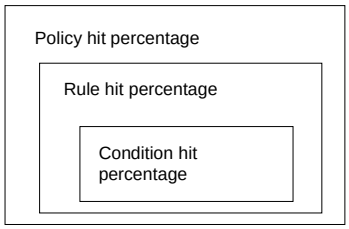

Figure 4.1: Policy Element Coverage

test criterion. The set of requests/test cases are usually redundant and could syntactically and semantically be the same. So, usually test case reduction techniques are used for removing these redundant cases.

There is a lot of research on the definition of test criteria and adequacy for programs [32]. The test criterion to be satisfied for a program is usually specified in the form of amount of coverage of the code. There are many coverage criteria like statement coverage, branch coverage, path coverage and mutation adequacy.

4.2 Policy Coverage Criteria

We have generated a set of requests for testing a policy. A policy test adequacy measure is needed to evaluate the quality of the requests in testing. We use the policy coverage criteria [15] as the adequacy measure for the requests generated by our approach. This policy coverage criteria defines the coverage similar to statement coverage in program testing. The criteria is to maximize the number of policy elements that a request covers. In other words, it can also be interpreted that this method choses those requests among a set of requests that cover the maximum number of policy elements. The policy elements granularity considered are Policy target element, Rule element and Condition element. They define the term “applicability” of the request to a policy element if a request satisfies the conditions for the policy element to evaluate to true. The measures used for coverage are, policy hit percentage, rule hit percentage and condition hit percentage. The policy hit percentage of a request is defined as the number of policies for which a request is applicable to the total number of policies. The rule hit percentage of a request is defined as the ratio of number of rules applicable to the request to the total number of rules. A rule hit requires that the request be applicable to the policy. The condition hit percentage of a request is the ratio of the sum of the conditions that evaluate to true and those conditions that evaluate to false to twice the number of

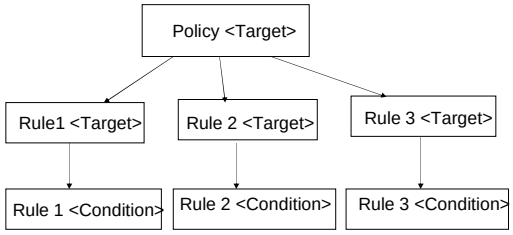

Figure 4.2: Target Driven Request Generation

total conditions. A condition hit requires that the request be applicable to the policy and the rule.

The figure 4.1 shows the above coverage criteria and the hierarchical relation between the various elements and its effect on the overall coverage.

We call the above coverage criteria as policy element coverage. The method that we have used in our testing framework converts the policy to program and uses the path coverage with concolic testing for limiting the number of paths. In our initial evaluation, we measure whether the requests generated by policy program path coverage achieve complete policy element coverage.

4.3 Target Driven Request Generation(Targen)

This is another method for policy request generation [13]. This method considers the policy as a hierarchical tree with each rule representing a leaf in the tree. The figure 4.2 shows this representation. Here, a request/test case is generated by solving constraints in the path of each rule from the root of the tree. The constraints in the path of a rule from the root of the tree has the target of the policy and the target of the rule to be solved. So, a request following this path is applicable to the policy and rule. Combinations of the attribute values in this path represent requests. Targen considers a modified form of combinatorial coverage that reduces the invalid requests generated from all possible combinations of the attribute values along this path. This request generation technique looks at each rule in the policy individually. It is possible that when all the rules in the policy are considered, some of the requests generated can be redundant. These redundant requests can be removed by measuring their coverage as explained in the next section.

4.4 Comparison of Request Generation Techniques

In this section, we compare the requests generated by the targen approach and that generated by our approach. We use the policy element coverage as the criteria for evaluating both the set

Table 4.1: Policies used in the evaluation.

policy	# set	# policy	# rule	# cond
codeA	5	2	2	0
codeB	11	5	5	0
codeC	8	4	4	0
codeD	11	5	5	0
default-2	1	13	13	12
demo-11	0	1	3	4
demo-26	0	1	2	2
demo-5	0	1	3	4
mod-fedora	1	12	12	10
simple-policy	1	2	2	0

of requests. In our experiments, we have used 10 XACML policies from [15]. These range from simple to complex policies. The table 4.1 shows the statistics of the policy composition. The first column gives the name of the policy, the second column gives number of policy sets in the policy, the third column gives the number of polices, the fourth column gives the number of rules in the policy and the fifth column gives the number of conditions in the policy.

The table 4.5 shows the coverage comparing the targen method and our approach using jCUTE. Column 1 gives the names of the policies. Columns 2, 3 and 4 give the policy, rule and condition coverage when using the targen approach. Columns 6,7 and 8 give the policy, rule and condition coverage when using the jcute based approach. The results show that each of the requests generated by our approach have 100% policy, rule and condition coverage. The targen method achieves 100% policy and rule coverage in the policy element coverage criteria. The condition coverage achieved is not 100% because the currently available targen tool does not implement conditions. Another difference to be noted is that the number of requests generated using the targen approach consists of some redundant cases - requests which do not cause any increase in the coverage. When using targen requests, a greedy reduction has to be performed to consider only requests which cause an increase in coverage. This is an extra overhead because first the request has to be generated and then the generated request has to be reduced. In our approach, however, the request generation process itself ensures that no request generated covers the same policy element as the previous request. This is because, in our approach only the constraints along the path to the rule are solved. Performing a greedy reduction of the requests generated by our approach showed that there was 0% reduction in the requests generated. This shows that our requests are optimal since they achieve 100% policy element coverage and no reduction is needed based on this measure.

The comparison of the two request generation techniques show that the both of the them

Table 4.2: Policy Set mutation operators

ID	Description
PSTT	Policy Set Target True
PSTF	Policy Set Target False
CPC	Change Policy Combining Algorithm

achieve almost 100% policy element coverage. But however, our approach to request generation considers the different paths in the policy program and seems like a semantically efficient measure than the targen approach. To check this, we use another test adequacy measure, mutation testing to compare the fault detection capability of both the request generation techniques. This technique is explained in the next section.

4.5 Mutation Testing for Fault Detection

We also use mutation testing to determine the quality of the requests generated. Mutation testing is used for testing programs by introducing small faults in the original program and generating programs that are close to the original program. Specifically, this technique takes advantage of the coupling effect [4]. In programming, the coupling effect can be defined on the basis of the empirical observation that complex errors occur due to the combination of simple errors. So, if we can introduce simple errors into a program by means of simple changes by using operators and if these errors can be detected by a test case, then we can be assured that this test case can also be used to detect complex errors which occur as a combination of these simple errors.

In other words, mutation testing measures the sensitivity of the test case to simple errors, which could be used as an indication of its sensitivity to complex errors. To use this technique for access control policies, it is necessary to identify simple errors in the context of an XACML policy. For example a simple error a user makes when writing an XACML policy is to write a policy with a set of rules but write a target which is not applicable to any valid request. Here the policy will not be applicable to any request. This can be emulated by creating a mutant policy with a target value that will always evaluate to false. Another mutant policy could be one with the target always being applicable, this will ensure that all the requests evaluate are applicable to the policy and the rules in the policy will be evaluated. Based on this idea, Martin and Xie [13] have developed set of mutation operators for an XACML access control policy.

The mutation operators can be classified based on the policy element on which the muta-

Table 4.3: Policy mutation operators

ID	Description
PTT	Policy Target True
PTF	Policy Target False
CRC	Change Rule Combining Algorithm

Table 4.4: Rule mutation operators

ID	Description
RTT	Rule Target True
RTF	Rule Target False
RCT	Rule Condition True
RCF	Rule Condition False
CRE	Change Rule Effect

tion operation is performed. They can be classified as,

Policy Set Mutation Operators: These represent the mutation operations that can be done at the policy set level. The various mutation operators defined for this is shown in table 4.2. They are, policy set target true mutant in which the policy set target is removed so that it is always true, policy set target false mutant in which the target value is changed such that it is always evaluated to false and change in policy combining algorithm mutant in which mutants are created for each policy combining algorithm like permit overrides, deny overrides and first applicable.

Policy Mutation Operators: These represent the mutation operations that can be done at the policy level. The various mutation operators defined for this is shown in table 4.3. They are policy target true, policy target false and change in rule combining algorithm. These are similar to the policy set operators except that the granularity is at the level of policy rather than policy set.

Rule Mutation Operators: These represent the mutation operations that can be done at the rule level. The various mutation operators defined for this is shown in table 4.4. They are rule target true, rule target false, rule condition true, rule condition false and change rule effect. The first two rule operators generate mutants by setting the rule targets to be true and false. The condition operators set the condition in each rule to be true and false. The change rule effect changes a rule with an effect of permit to one with an effect of deny and vice versa.

Table 4.5: Policy coverage and fault detection when using targen and the jcute technique.

	targen				jcute			
policy	pol %	rule %	con %	mut kill%	pol %	rule %	con %	mut kill%
codeA	100	100	n/a	36.36	100	100	n/a	41.8
codeB	100	100	n/a	37.7	100	100	n/a	38.58
codeC	100	100	n/a	38.58	100	100	n/a	38.54
codeD	100	100	n/a	37.79	100	100	n/a	37.79
default-2	100	100	100	50	100	100	100	31.6
demo-11	100	100	75	77.78	100	100	100	88.88
demo-26	100	100	50	78.57	100	100	100	78.57
demo-5	100	100	75	78.95	100	100	100	89.47
mod-fedora	100	100	100	56.67	100	100	100	44.16
simple-policy	100	100	n/a	44.44	100	100	n/a	55.5

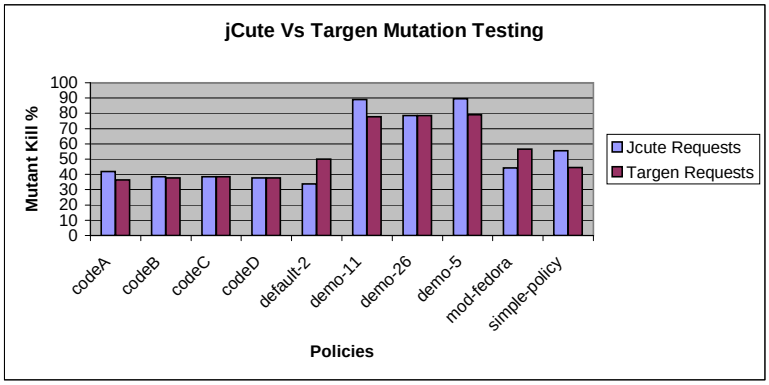

Figure 4.3: jCute Vs Targen

The number of mutants created for each policy element is dependent on the size of the policy. For example, the number of policy set mutants with a target of a true is equal to the number of policy set target elements within a policy. Some of mutants may also be equivalent and this is dependent on the specific policy.

These mutation operators represent some of the possible changes that can be introduced in the policy. The requests generated by our approach are evaluated against these mutant policies. If the result of the evaluation is different than the original policy, the mutant is said to have been killed.

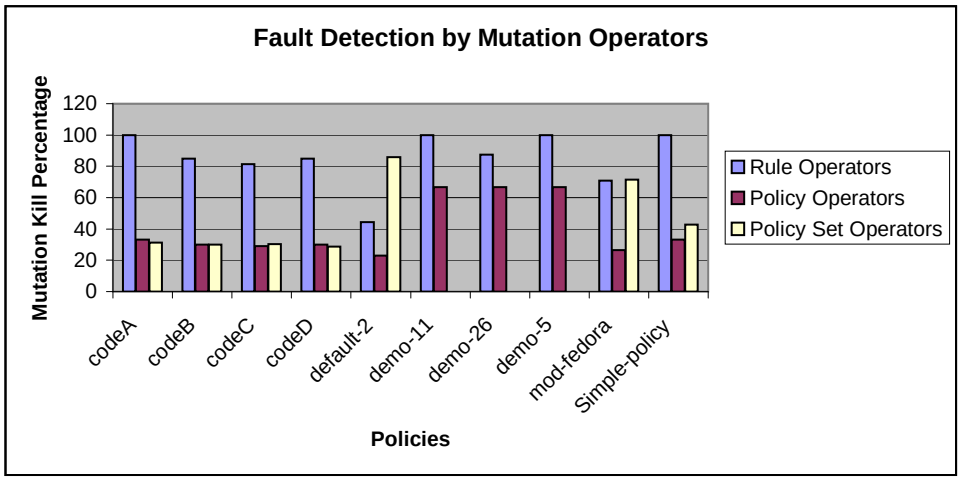

Figure 4.4: Mutation Operators

4.5.1 Fault Detection Capability Comparison

We compare the fault detection capability of our method and the target-driven request generation method. The set of requests generated by both the methods are evaluated against the original policy and the mutant policies. If the results are different, the mutant is said to have been killed. If the result remains the same, the mutant lives. The column 5 in table 4.5 shows the mutation kill percentage of the requests from targen and the column 9 shows the mutation kill percentage using our approach. The figure 4.3 shows the graph representation of the comparison. The fault detection capability of jCute based optimal requests performs better or as good as targen in most of the cases. However, in two of the cases targen performs better than jcute. The following shows an example of a case where a mutant is killed by targen but not killed by jcute.

The original policy set has the following two policies among a set of policies,

Policy 2 :Deny access to *POLICY* resource unless the subject has *administrator* role.

Policy 8 :If subject has *administrator* role, access decision is Permit. (This broad rule may be limited by specific rules in the beginning.)

The XACML representation of this policy is shown in figure 4.5. Consider the mutation operator, Change Rule Effect(CRE) which generates a mutant of the policy by changing the effect of the rule in the Policy 2 to permit. So, the original policy denies access to *POLICY* resource if subject does not have *administrator* role and the mutant policy permits access if subject has *administrator* role.

Among the set of requests generated by targen, a request with just the attribute *POLICY* is generated. This will give an effect of deny in the original policy, being applicable to Policy 2 and an effect of permit in the mutant policy again being applicable to Policy 2.

Among the Jcute requests, the request being applicable to Policy 2 has the attributes *POLICY* and *administrator*. For this request, the result in both original and mutant policy is permit (being applicable to Policy 8 in both cases). So the mutant is reported as not killed. Here a request with only *POLICY* attribute is not generated.

It has to be noted that jCute generates an optimal set of requests that achieve the maximum path or branch coverage. So, even if a request with only *POLICY* attribute is generated, it may not be chosen as an optimal request because another request with both *POLICY* and *administrator* attributes achieves more coverage. Similarly for other cases also some requests with attribute values covering multiple paths may be reported by jCute as an optimal request causing some mutants to be missed when doing mutation testing.

We analyzed the effect of each type of mutation operators on the fault detection capability in the jcute case. The figure 4.4 shows the graph with the mutation kill percentage for each of the operators. The higher the fault detection percentage value, the better because it means more mutants are killed. It is observed that the rule operators have better fault detection than policy and policy set operators for almost all the policies.

It is to be noted that some of the mutants cannot be killed because they could be equivalent mutants [17]. An equivalent mutant is one which is semantically equivalent to the original policy. An example of an equivalent policy mutant is a mutant create by changing the policy set combining algorithm from first applicable to permit overrides when there is only one policy in the policy set. In this case, both the mutant and the original policy are semantically the same. In general, it is difficult to automatically detect equivalent mutants and is often done manually [17]. The mutation testing tool that we use does not detect and remove equivalent mutants.

For the policies default-2 and mod-fedora for which the mutant kill percentage of jcute is lesser than targen, we measure the combined mutant kill%. To evaluate if the same mutants are killed in both the cases, we combine the set of requests generated from both the approaches and measure the mutant kill %. The combined mutation kill % is higher than the individual total mutant kill %. This show that the mutants killed by jcute are different than that killed by jcute. The graph is shown figure 4.6.

The figure 4.7 shows the comparison of the number of requests generated by targen and jcute. It is observed that the number of requests generated by targen is dependent on the number

of rules in the policy while the number of requests generated by jcute is those set of requests that achieve optimal path coverage in the policy program.

Using only the existing policy coverage criteria, the requests generated by both targen and jcute methods have the good structural coverage. The targen set of requests, though they achieve 100% coverage most of the time, they look at each rule in the policy locally, while a path coverage based set of requests will look at all the rules in the policy when formulating the set of requests. The difference between the two request generation techniques can be better observed by introducing a new policy coverage measure called the *policy path coverage*.

4.6 Policy Path Coverage

'A test adequacy criteria is a predicate that defines what properties of a program must be exercised to constitute a through test' [32]. To define the test adequacy criteria for the policy, the property of the policy has to be analyzed. The property that we are considering here is the path coverage of the various requests. We define the policy path coverage measure based on the execution path coverage of the policy program.

For a program, the path coverage criteria requires that all the execution paths from the program's entry to its exit are executed during testing [32]. The corresponding policy path coverage can be similarly defined as,

policy path coverage : The policy path coverage criteria requires that all the valid policy evaluation paths covering the first rule to the last rule in the policy must be covered at least once.

Examples of valid policy paths are, the outermost path in a policy which evaluates the default case in the policy. The innermost path in a policy is the one for which only the last rule in the policy is true. This means all the previous rules evaluate to false.

The other components in an XACML policy are policy set which can contain policies and policy sets. In our conversion from policy to program code, we order the policy set components in the order in which they appear in the policy. Also in XACML, two combining algorithms, deny overrides and permit overrides specify the order in which the rules have to be combined. These algorithms are defined to given importance to rules with an effect of Permit(Deny) when multiple rules evaluate to true. We convert these into first applicable format and so the path coverage is the same as the case for first applicable.

The target-driven requests are not expected to achieve complete policy path coverage. An analysis of the way these requests are generated gives an indication of this. In their method, each rule is treated separately and they use a modified form of combinatorial coverage in which combinations of requests with subject, resource and action attributes in every rule are formulated. Among these large number of generated requests, those causing an increase in the policy structural criteria are chosen as the final set of requests. These requests do not consider the interaction between the rules in the policy. So these requests may not achieve complete policy path coverage.

4.7 Threats to Validity

The extend to which the example policies, mutation operators, coverage metrics and requests sets truly reflect actual practice has an effect on the external validity. More mutation operators are needed at a lower level to test other aspects of a policy. The internal validity is threatened by faults in our implementation of the conversion tool as well as faults in the tools that we use for evaluating our method.

```
<Policy PolicyId="MyPolicySet.2" RuleCombiningAlgId="first-applicable">
  <Description>deny access to POLICY datastream unless subject
    has administrator role</Description>
   <Target>
     <Resources>
       <Resource>
         <ResourceMatch MatchId="function:string-equal">
         <AttributeValue>POLICY</AttributeValue>
         <ResourceAttributeDesignator
         AttributeId="resource:datastream:id"/>
          </ResourceMatch>
        </Resource>
      </Resources>
    </Target>
    <Rule RuleId="MyPolicySet.2.r.1" Effect="Deny">
      <Condition FunctionId="function:not">
        <Apply FunctionId="function:string-is-in">
          <AttributeValue>administrator</AttributeValue>
          <SubjectAttributeDesignator AttributeId="fedoraRole" />
        </Apply>
      </Condition>
    </Rule>
  </Policy>
  <Policy PolicyId="MyPolicySet.8"
  RuleCombiningAlgId="first-applicable">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="function:string-equal">
            <AttributeValue>administrator</AttributeValue>
            <SubjectAttributeDesignator  AttributeId="fedoraRole" />
          </SubjectMatch>
        </Subject>
      </Subjects>
    </Target>
    <Rule RuleId="MyPolicySet.8.r.1" Effect="Permit"/>
  </Policy>
```

Figure 4.5: Example : Original Policy

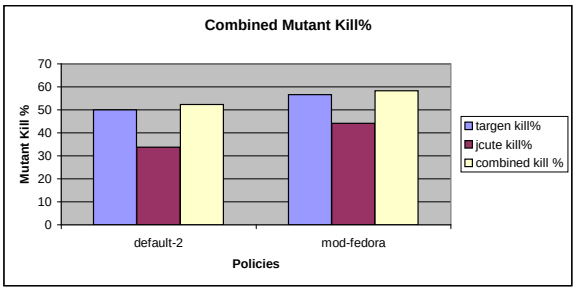

Figure 4.6: Combined Mutation Kill Percentage

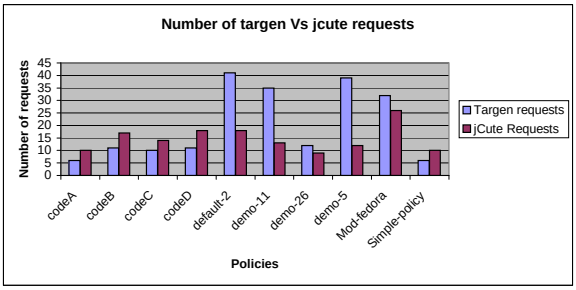

Figure 4.7: Comparison of Number of Requests

Chapter 5

Related Work

In this chapter, we discuss work related to access control models, polices and also other techniques that are used for analyzing access control polices.

5.1 Policies, Models and Mechanisms

Any system implementing access control must consider the three abstractions [20]:

1. Security Policy: This defines high level rules according to which access to resources and data within a system will be granted or denied. An example of a security policy used at a school could be, the TA can assign only internal grades.

2. Security Model: This gives a formal representation of how the access control security policy is implemented in the system. This can be used to give a proof of the properties provided by the system. It can be said that the model bridges the gap in abstraction between policy and mechanism [27]. An example security model is the mandatory access control model, where the level of access of an entity depends on the security clearance level assigned to it like top secret, secret,normal.

3. Security Mechanism: This defines the actual system specific functions that implement the controls imposed by the policy and formally stated in the model. An example security mechanism is access control lists.

5.2 Access control models

Access control models are grouped into three main classes : discretionary model, mandatory model and role based model. Our approach to policy testing can be applied to all policies build on any of these models.

Discretionary policy model

In discretionary access control, a list of authorizations are specified for each subject in the system. The system gives access to a subject by looking up whether a subject has access to an object in the authorizations specified. Different subjects can have different levels of access to one object. In this model, the users have the discretion of granting or revoking privileges to other users. The access matrix model is used for describing discretionary access control. In access matrix, the rows are the subjects in the system and the columns are resources to which a subject's access has to be controlled. The cell intersecting the row and column will specify the access level of the subject to the resource. This matrix model can be implemented as,

authorization table : Here the authorizations are represented as a table. This is mostly used in databases by creating a table with columns subject, resource and action. Each entry in the table represents an authorization.

access control list : In an access control list implementation, every column in the access matrix is a list.(i.e) There is a list for each object in the system specifying the subjects that have access to that particular object.

capability list : In a capability list implementation, every row in the access matrix is a list. (i.e) There is a list for each subject in the system specifying the different objects that the particular object has access.

Each implementation has its own advantages and disadvantages and a particular implementation is chosen depending on the needs of the specific application. Discretionary policies however are not secure against attacks from the processes invoked by legitimate users that may perform malicious functions on behalf of the user. An example of this vulnerability is a trojan horse program that is executed by a subject like a high level user that reads from one sensitive file and writes to another common file to which a lower level user has read access. Now the low level user will be able to read the contents of the sensitive file.

Mandatory policy model

Mandatory policies classify the subjects and objects within the system into different security clearance levels. The various mandatory policies based on the semantics of the classification are,

Secrecy-based mandatory policies These control the 'direct and indirect flow of information to the purpose of preventing leakages to unauthorized subjects' [20]. Users can connect to the system at different levels and the two Bell La Pendula principles to be satisfied are: No-read-up and No-write-down. Enforcing this restriction ensures that no information flow exists from one level to another.

Integrity based mandatory policy The Biba model protects the integrity of a resource. The integrity classification reflects the trustworthiness of the user in modifying the information and for an objects it refers to the trustworthiness placed on the data provided by the system. Access control is enforced by the following two principles: No-read-down and no-write-up. Enforcing this principle safeguards the integrity by ensuring that objects at a lower level which are less reliable cannot write to levels above it.

Hence, secrecy policies allow the flow of information from lower to higher secrecy classes while integrity policies allow the flow of information from higher to lower integrity classes. So to ensure both secrecy and integrity both the classes must be defined.

Though mandatory policies provide protection against information leakages, they cannot guarantee complete secrecy because they do not offer protection from covert channel communication.

Mandatory and discretionary polices are combined and the chinese wall policy model is defined. This policy model was proposed to enforce the mandatory control on discretionary policy implementations found in commercial systems. It combines mandatory and discretionary policies. The classification class restricting the information flow here reflects the flow of information between conflicting business classes for an individual consultant. Here, access to data is not constrained by its classification but by what data a subject has already accessed. Though this policy has some limitations of mandatory policies like being rigid in a commercial setting, this is a good example of applying 'dynamic separation of duty constraints present in the real world and has been taken as a reference in building subsequent policies and models' [20].

Other work combining discretionary and mandatory access control include authorization based information flow policies. Also, discretionary policies have been modified for expanding authorizations to support conditions in the policy. Also, authorizations can be extended with temporal constraints.

Another aspect of access control is the administrative policies which specify who is authorized to manage the access rules and decisions. In mandatory, there must be a centralized authority specifying the security class of the objects. In the case of discretionary, there can be different subjects like, centralized, hierarchical, cooperative, ownership and decentralized.

Role based policy model

Role based access control(RBAC) [21] specify access based on what roles the users of the system assume. RBAC defines users, roles and permissions. Each role is associated with some permissions and users can assume different roles. This model is non-discretionary and is best suited in an enterprise environment where the users of the system change frequently while the roles remain the same. RBAC removes the rigidity of MAC and also adds on to the security of DAC. So it can be thought of as a policy model combining the advantages of the previous two approaches. Another advantages of using RBAC is that RBAC itself can be used for administration of RBAC policies.

5.3 Policy Specification Langauges

Our policy testing technique can be applied to other policy specification languages also. Here, we describe some of the common policy specification languages.

5.3.1 Ponder Policy Specification Language

Ponder [3] was developed as part of an academic project at Imperial College in London. Ponder is a declarative object oriented policy specification language. It is more suitable for access control enforcement in distributed and network systems. They separate policy from implementation and enable dynamic management of the policies. The key terms are,

Subject : Subject refers to users or principles or any other automated entity which has a management responsibility.

Target : Target refers to resources or services in the system.

```
Inst(auth+ | auth-)policyname
Subject domain-scope-expression;
Target domain-scope-expression;
Action domain-scope-expression;
[When  constraint-expression]
```

Figure 5.1: Ponder Authorization policy syntax

```
Inst(auth+)policyname
Subject faculty;
Target grades;
Action Assign, View;

//This policy authorizes faculty to assign and view grades.
```

Figure 5.2: Ponder Authorization policy example

Domains : Domains provide a way for grouping subjects or targets.

Ponder specifies the following types of polices for expressing access control,

Authorization policies : These are the access control policies specifying what targets a subject can access. The policy can express both positive and negative authorizations. The positive authorization policies specify what actions a subject can perform while negative authorization specifies those actions a subject is forbidden from performing. The figure 5.1 gives the syntax of the authorization policy.

The university policy can be represented in the figure 5.2 as,

Information filtering policies : These policies place restrictions on the actions performed. They can be used to provide an additional level of restriction in addition to an authorization policy that grants an action.

Delegation polices : This policy enables one user to delegate access rights to another user.

Refrain policies : Refrain policies define the actions that subjects must not perform on target objects even though they may actually be permitted to perform the action. They are similar to negative authorization policies but are enforced on the target rather than on the subject.

Obligation policies : These policies specify the actions that need to be performed by managers when certain events occur within the system.

Ponder also supports various constraints like basic policy constrains and meta-policy constraint. Basic policy constraints are expressed in terms of a predicate which has to evaluate to true for the policy to apply. Meta-policies are used to specify policies about policy and the constraints are on self management and separation of duty. With all the above features, a large enterprise can structure its access control policy. Ponder also provides other features to enable the ease of management of large complex policies. We can specify groups for packaging related policies, roles for semantic grouping of policies with common subjects. Also, they support policy hierarchies and the policy types can be specialized and reused. Relationships can also be defined showing the definition of roles participating in interactions.

They also enable the specification of management structures which is a composite policy containing the definition of roles, relationships and other nested management structures as well. This structure can be defined in general for a branch of company or a department of a university. This can then be instantiated for particular departments or departments.

Testing of ponder using our approach is natural. For a given composite ponder policy can be converted to a java class. All the different policy types can be defined as methods. The interaction between the methods can be easily captured in the concolic testing approach.

5.3.2 The Platform for Privacy Preferences(P3P)

The Platform for Privacy Preferences (P3P) is a specification from the World Wide Web Consortium (W3C) for specifying the privacy policies of enterprises. Though the specification is platform independent and can be used across enterprises, it is not a general purpose specification. The P3P policies are higher level policies usually published by an enterprise to reveal their privacy practices to customers.

5.3.3 Enterprise Privacy Authorization Language(EPAL)

Enterprise Privacy Authorization Language(EPAL) [18] was developed at International Business Machines(IBM). It is submitted for review to W3C. EPAL is mainly designed as a privacy policy interoperability language suitable for exchange between enterprises in a structured format. The language is appropriate for representing the data-handling practices and policies within and between enterprises that want to have a systematic way of managing privacy. This is also useful for automatic audit control of the accesses to the information and also for enforcing accountability of privacy practices.

```
 <rule id=‘‘univ-policy’’ ruling=‘‘allow’’>
<user-category refid=‘‘faculty’’ />
<data-category refid=‘‘student-information’’ />
<purpose refid=‘‘view-and-assign-grades’’ />
<action refid=‘‘view, assign’’ />
<condition refid=‘‘condition’’ />
</rule>
```

Figure 5.3: EPAL policy example

EPAL defines the attributes as a list of hierarchies of,

data-categories : This specifies the different ways in which the different data collected by an enterprise is used depending on the sensitivity of the data. For example, the medical-record data is more sensitive than the contact information.

user categories : This categorises the different users of the data. In the above example, the medical record information is used by the doctor and the contact information is used by the sales department.

purposes : This specifies the purpose for which the categorized data is used by the categorized user. The doctor will use the medical record for purpose of scheduling tests and the sales department will use the contact information for shipping purposes.

They also define actions, obligations and conditions. Actions specify how the data is used, obligations specify what must be satisfied in the environment and conditions must evaluate to true in the context for the rule to be applicable. An EPAL policy is a list of rules that are ordered according to descending precedence.

The figure 5.2 gives the example for an EPAL policy

A study comparing XACML and EPAL concludes that EPAL uses a lot of XACML and that EPAL is a subset of XACML except for some specific features. For instance, EPAL and XACML share the same framework of a policy made up of a series of rules. A rule is applicable only if the condition in it evaluates to true and the effect of the rule is returned. Also, both languages share the same framework for the requests: a request is made up of a collection of attribute values.

5.4 Policy Testing Techniques

Martin et al [15] have developed a systematic method for testing access control policies. Theirs is the first work on defining and measuring structural coverage of access control policies for testing. They have developed a coverage measurement tool for measuring policy coverage given a set of XACML policies and set of requests. Their coverage criteria is based on the structure of the policies and is similar to statement coverage in a program. The request generation process is random and the requests are got by setting bits in a vector of policy attribute values. Even though the random request generation technique does not repeat requests that are already generated, this method has the disadvantage of using the random test input selection strategy. They use a tool to greedily reduce requests from the generated set of requests based on the coverage measure. They also perform mutation testing to analyze the fault detection capability of the reduced set of requests. [13] uses combinatorial coverage and considers the policy as a hierarchical tree with each rule representing a leaf in the tree. Here, a test case is generated by solving constraints in the path of the rule from the root of the tree. Even though this method achieves better coverage than random request generation, this request generation technique looks at each rule in the policy individually only and the entire effect of the sequence of constraints in the policy is not considered. In our approach, the ordering of the rules in the entire policy path is considered and the constraints along the path are solved. Hence, our method provides a better measure of the coverage.

Another area where access control policy testing is done is firewalls protecting network resources. Al-Shaer et al [5] propose automated testing of firewalls with respect to their internal implementation and security policies. They propose a novel firewall testing technique using policy-based segmentation of the traffic address space, which can intelligently adapt the test traffic generation to target potential erroneous regions in the firewall input space. Though this method is efficient, it is applicable only to firewall polices because they have made the testing dependent on the structure of the access control policy in a firewall. However, the idea of analyzing the logs of packets/request cannot be applied as such to any general purpose access control system.

5.5 Formal Policy Analysis

A complementary approach to access control policy testing is to convert the policy to a logical representation and use formal analysis techniques for verification and analysis. Hughes and Bultan [9] translated XACML policies to their logical representation in the Alloy language and

checked their properties using the Alloy Analyzer. Using their translator and the Alloy analyzer, it is possible to check a policy which is implemented as a combination of sub-polices correctly reproduces the properties of the sub policies. This approach, though produces good results does not scale well with increase in the size of the policy. Zhang et al [31] propose a mechanism for evaluating XACML polices through model checking. They evaluate whether the policies give legitimate users enough permissions to reach their goals and also to check whether the policies prevent intruders from reaching their malicious goals. However, the access control polices have to be translated to the RW language to apply their techniques. The limitations of these above approaches is that they do not treat all the features of XACML. Also, a predefined set of properties about the policy should be given which, does not exist in practice. Also, this analysis can become intractable when there are more attributes in the policy. The advantage of using testing is that no translation to a separate domain is needed to check the policies. Also, all features of XACML can be tested.

Margrave [6] is an efficient tool that enables checking for semantic inconsistencies in the policy and returns counter examples representing cases which are causing violation of properties of the policy. Change impact analysis is done between two policies to determine the properties of the policy. They construct a multi-terminal binary decision diagram to represent the rules in the policy. However this tool does not support all features of XACML.

We have defined a framework for testing access control policies by converting them into programs and using a restricted form of path coverage criteria. Other software applications like database applications and grammar-based software also have specific criteria for their testing. Hennessy and Power [8] propose a strategy for the construction of test suites for grammar based software. The reduction criterion they use is based on the rule coverage of the test suites. They analyze if the code coverage and fault detection capability are reduced because of the reduced test suite. Suarez-Cabal and Tuya [26] have developed a tool for the automated testing of SQL queries. They define a coverage tree for the different condition branches in the SQL SELECT statement. Kapfhammer and Soffa [12] define a framework for testing database driven applications and the control flow between various entities in such an application. They define the test adequacy criteria for the database application based on the database interaction flow graph showing the interaction between the various entities.

Chapter 6

Conclusions and Future Work

Policy testing is a practical technique for the quality assurance of access control policies. In this thesis, we propose a method that uses *policy programs* for testing access control policies. An automated tool has been developed based on this method. Given a policy, our tool can generate an optimal number of requests for testing the policy. The advantage of our approach is that, we use existing software testing techniques that are being used for testing different software applications. Also, our approach is general and can be applied for testing most rule based policy specifications even in other languages. We have automated the entire testing process, so, any changes made to the policy can be easily tested. This is particularly useful in an enterprise environment where the policies are large and are also revised over a period of time.

We evaluate our method by testing with ten XACML policies. The test cases generated by our method achieve the complete XACML policy structural coverage which is the existing adequacy criteria for testing an XACML policy. We perform mutation testing on the policy and the generated request set and compare our results with the other existing techniques. The mutant kill percentage is as good as or better than existing techniques in most of the cases. Also, the results indicate that the mutants created by the rule operator have more kill percentage than that achieved by other operators. This shows that the use of the policy program for generating test cases is able to capture fine errors created by mutants. Based on this, we have motivated the definition of a policy coverage criteria based on our approach to policy testing. This criteria is expected to be stronger than the existing policy structural coverage. The stronger the coverage criteria, the better will be the quality of the test cases generated. In future, a coverage tool can be developed based on this criteria. This tool can act as a stronger test adequacy measure for policy testing.

In future, program analysis on the policy program can be done to semantically analyze the

policies. Also, change impact analysis can be done between two policy programs. The difference in the set of test cases returned between a policy program and a changed policy program can be analyzed. Also, we can perform mutation testing at the program level and compare the results. For this, a new set of operators should be defined at the program level. However, it should be ensured that the a program mutant maps to a policy mutant. Tools available for performing mutation of a program can be used to create other different types of mutants and they can be analyzed. It would be interesting to analyze if the mutation operators at the program level correctly represent the common user errors done at the policy level.

Bibliography

[1] R. J. Anderson. A security policy model for clinical information systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 30–43, 1996.

[2] C. Bussler and S. Jablonski. Policy resolution for workflow management systems. In *Proc. Hawaii International Conference on System Science*, Maui, Hawaii, January 1995.

[3] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proc. International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, 2001.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[5] A. El-Atawy, K. brahim, H.Hamed, and E. Al-Shaer. Policy segmentation for intelligent firewall testing. In *1st IEEE ICNP Workshop on Secure Network Protocols, 2005. (NPSec)*, pages 67–72, Nov 2005.

[6] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proc. 27th International Conference on Software Engineering*, pages 196–205, 2005.

[7] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[8] M. Hennessy and J. F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 104–113, November 2005.

[9] G. Hughes and T. Bultan. Automated verification of access control policies. Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara, 2004.

[10] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proc. 1997 IEEE Symposium on Security and Privacy*, pages 31–42, 1997.

[11] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 474–485, 1997.

[12] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proc. 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–107, 2003.

[13] E. Martin and T. Xie. Automated mutation testing of access control policies. Technical Report TR-2006-12, Department of Computer Science, North Carolina State University, Raleigh, North Carolina, 2006.

[14] E. Martin and T. Xie. Automated test generation for access control policies. In *Supplemental Proceedings of the 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, November 2006.

[15] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. 8th International Conference on Information and Communications Security (ICICS 2006)*, pages 139–158, December 2006.

[16] OASIS. OASIS eXtensible Access Control Markup Language (XACML). `http://www.oasis-open.org/committees/xacml/`, 2005.

[17] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, October 2000.

[18] P.Ashley, S.Hada, G.Karjoth, C.Powers, and M.Schunter. Enterprise Privacy Authorization Language (EPAL). `http://www.w3.org/Submission/EPAL/`, 2003.

[19] T. Ryutov and C. Neuman. Representation and evaluation of security policies for distributed system services. In *Proc. DARPA Information Survivability Conference and Exposition*, January 2000.

[20] P. Samarati and S. D. C. di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD '00: Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design*, pages 137–196, London, UK, 2001. Springer-Verlag.

[21] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[22] A. Schaad, J. Mo, and e Jacob. The role-based access control system of a european bank: A case study and discussion, 2001.

[23] K. Sen. Scalable automated methods for dynamic program analysis. In *PhD Dissertation*, 2006.

[24] K. Sen and G. Agha. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In *Proc. 18th International Conference on Computer Aided Verification*, pages 419–423, 2006. (Tool Paper).

[25] E. Sirer and K. Wang. An access control language for web services. In *Proc. 7th ACM Symposium on Access Control Models and Technologies*, Monterey, CA, June 2002.

[26] M. J. Suarez-Cabal and J. Tuya. Using an SQL coverage measurement for testing database applications. In *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2004.

[27] V.Hu, D.Ferraiolo, and R. Kuhn. Assessment of access control systems. NISTIR, Sept. 2006.

[28] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Terminology for policy-based management. RFC 3198 (Informational), Nov. 2001.

[29] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, 2004.

[30] R. Yavatkar, D. Pendarakis, and R. Guerin. A framework for policy-based admission control. RFC 2753 (Informational), Jan. 2000.

[31] N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In *Proc. 8th International Conference on Information Security*, pages 446–460, September 2005.

[32] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Appendix

Appendix A

Fedora XACML Policy Example

```
<PolicySet PolicySetId="MyPolicySet"
PolicyCombiningAlgId="first-applicable">
  <Target/>
  <Policy PolicyId="MyPolicySet.0" RuleCombiningAlgId="deny-overrides">
    <Description>deny any access if client ip address is not 127.0.0.1
    </Description>
    <Target>
      <Actions>
        <Action> urn:fedora:names:fedora:2.1:action:api-m </Action>
      </Actions>
    </Target>


    <Rule RuleId="MyPolicySet.0.r.1" Effect="Deny">
      <Condition FunctionId="function:not">
       <AttributeValue>127.0.0.1</AttributeValue>
      </Condition>
    </Rule>
<Rule RuleId="MyPolicySet.0.r.2" Effect="Deny">
  <Description>deny any access to objects or data streams,
  which are either inactive or deleted,unless subject has
  administrator role</Description>
    <Target> <Resources>
       <Resource>Inactive_object</Resource>
        <Resource>Deleted_object</Resource>
        <Resource>Inactive_datastream</Resource>
        <Resource>Deleted_datastream</Resource>
      </Resources> </Target>
         <Condition FunctionId="function:not">
          <AttributeValue >administrator</AttributeValue></Condition>
    </Rule>
<Rule RuleId="MyPolicySet.0.r.3" Effect="Permit">
<Target><Subject>special_user</Subject>
<Resource>Deleted_datastream</Resource></Target>
</Rule>
</Policy>
</PolicySet>
```

Figure A.1: Fedora example XACML policy