Automated Testing for RBAC Policies

by

Poonam Gupta

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2014 by the
Graduate Supervisory Committee:

Gail-Joon Ahn, Chair
James Collofello
Dijiang Huang

ARIZONA STATE UNIVERSITY

May 2014

ABSTRACT

Access control is necessary for information assurance in many of today's applications such as banking and electronic health record. Access control breaches are critical security problems that can result from unintended and improper implementation of security policies. Security testing can help identify security vulnerabilities early and avoid unexpected expensive cost in handling breaches for security architects and security engineers. The process of security testing which involves creating tests that effectively examine vulnerabilities is a challenging task.

Role-Based Access Control (RBAC) has been widely adopted to support fine-grained access control. However, in practice, due to its complexity including role management, role hierarchy with hundreds of roles, and their associated privileges and users, systematically testing RBAC systems is crucial to ensure the security in various domains ranging from cyber-infrastructure to mission-critical applications.

In this thesis, we introduce i) a security testing technique for RBAC systems considering the principle of maximum privileges, the structure of the role hierarchy, and a new security test coverage criterion; ii) a MTBDD (Multi-Terminal Binary Decision Diagram) based representation of RBAC security policy including RHMTBDD (Role Hierarchy MTBDD) to efficiently generate effective positive and negative security test cases; and iii) a security testing framework which takes an XACML-based RBAC security policy as an input, parses it into a RHMTBDD representation and then generates positive and negative test cases. We also demonstrate the efficacy of our approach through case studies.

DEDICATION

To my beloved parents

and in loving memory of my in-laws.

ACKNOWLEDGEMENTS

First, I would like to convey my sincere thanks to my advisor Professor Gail-Joon Ahn. He has advised and guided me throughout my thesis journey. Second, I would like to convey my sincere thanks to my supervisory thesis committee Professor James Collofello and Professor Dijiang Huang. And special thanks to Carlos Rubio Medrano for helping me with all aspects of the thesis.

I have been fortunate to have multiple internships at Intel Corp., Chandler, which provided me opportunity to apply the knowledge I learned through course work at ASU in an industry setting. The internships also enabled me to learn various other skills necessary for pursuing a professional career.

Last but not the least I would like to convey my thanks to my family for their unconditional love and support.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

1.1 Approach 4

1.2 Contributions 7

1.3 Thesis Outline 8

2 BACKGROUND AND RELATED WORK 9

2.1 Multi-Terminal Binary Decision Diagram (MTBDD) 9

2.2 JUnit 11

2.3 Role-Based Access Control (RBAC) 11

2.4 eXtensible Access Control XML (XACML) 12

2.5 Security Module/Unit Testing 14

2.6 Related Work 14

3 SECURITY TESTING APPROACH 18

3.1 Preliminaries 18

3.1.1 RBAC Model 18

CHAPTER Page

3.1.2 Goal of Security Testing 19

3.1.3 Positive and Negative Test Cases 20

3.2 Security Testing Approach for RBAC 21

3.2.1 Generating tests for single authorized role 21

3.2.2 Optimizing tests for multiple authorized roles 22

3.2.3 Optimizing Negative Test Cases 24

3.3 Role Hierarchy MTBDD (RHMTBDD) 26

3.3.1 Introduction of The Decision Node 29

3.3.2 Structure of RHMTBDD 31

4 SECURITY TESTING FRAMEWORK: IMPLEMENTATION DETAILS AND EVALUATION 33

4.1 Security Testing Framework 33

4.1.1 XACML Policy Parser 33

4.1.2 Generation of Test Cases from RHMTBDD 37

4.1.3 Approaches for Creating Positive Test Cases 39

4.1.4 Approach for Creating Negative Test Cases 40

4.2 Summary of Framework Implementation 40

4.3 Execution of security test cases 41

4.4 Evaluation 43

4.4.1 Evaluation of Maximum Privileges 43

4.4.2 Evaluation of Fronts Coverage Approach 46

5 CONCLUSION AND FUTURE WORK 49

5.1 Conclusions 49

CHAPTER Page

5.2 Future Work 49

5.2.1 Static Separation of Duty (SoD) 49

5.2.2 Integration with JET 51

REFERENCES 52

LIST OF TABLES

Table Page

Table 1: Unique and Inherited Action for Figure 9a) and Figure 9b)............................... 27

Table 2: Example of Positive Test Cases... 39

Table 3: Example of Negative Test Cases 40

LIST OF FIGURES

Figure	Page
Figure 1: Policy Representation in MTBDD.	10
Figure 2: Combining PMTBDDs to get PMTBDD for the aggregate policy.	10
Figure 3 : RBAC Policy in XACML [2].	13
Figure 4: Positive and Negative Test Case for |auth(r)| = 1.	22
Figure 5: An example role hierarchy.	23
Figure 6: Optimizing test cases for mutual non-dominating role.	23
Figure 7: Reducing Negative Test Cases.	25
Figure 8 : Fronts Coverage Criteria.	26
Figure 9: Hierarchical RBAC examples.	27
Figure 10: RHMTBDD for Figure 9 (a).	29
Figure 11: RHMTBDD for Figure 9 (b).	30
Figure 12: Implementation framework for creating Test Cases.	33
Figure 13 : Parsing XACML Policy.	34
Figure 14 : Output of XACML Policy Parsing.	35
Figure 15: Security Test Cases.	42
Figure 16: Results of Security Test Cases.	43
Figure 17: Evaluation Results.	46
Figure 18: Example of a) role hierarchy for front coverage and b) the corresponding unique actions associated with each role in the hierarchy.	47
Figure 19: Results of FCC for a) Teller role and b) Customer role; and c) Over all the roles for various k values.	47
Figure 20: Representing Complex Security Policy.	50

CHAPTER 1
INTRODUCTION

As technology is getting more and more sophisticated and connected, security is becoming an increasingly greater concern. Cyber systems in almost every domain including commercial, medical, and entertainment need to overcome adverse activities from various malicious entities. Security policy is defined to accommodate the security needs for a system or an infrastructure. Security policy specifies security properties needed to be satisfied for a system. Traditional security properties include confidentiality, integrity, and availability (CIA) properties as well as usage property [1]. For example, a simple security property to ensure confidentiality could be that no senior-level person can write to a junior-level resource and no junior-level person can read a senior-level resource [1]. Such and more complex security properties can be specified using security policy languages such as XACML [2]. In addition, policy management is one of important security mechanisms to check the assurance of the specified security policies and enforce those policies.

Several security mechanisms such as biometrics and crypto primitives have been developed to accomplish the required security properties [1]. Access control mechanisms are essential to accomplish many of the security properties including confidentiality. Two basic access control models are: *discretionary access control* (DAC) and *mandatory access control* (MAC) [1]. DAC based mechanism have been used in various operating systems and data bases systems but it is hard to manage since this mechanism is based on users' complex intentions, whereas MAC based mechanisms are very common in the military domain which is not applicable to most computing environments. To address inherent limitations in both the mechanisms, another access control model called *Role based access control* (RBAC) [3] was introduced and has become popular since 1) it is better aligned with how businesses operate – people are assigned various roles with specific privileges in the companies' appropriate hierarchy and restrictions are put in place such as *separation of duty* (SoD) so that no single person can be over-privileged to make any severe harm to the company; and 2) the generality of RBAC can help enforce both DAC and MAC based mechanisms [3]. However, its flexibility can be problematic in ensuring access control requirements especially for large-scale companies with complex, and dynamic role hierarchies and constraints. Hence, automatic testing techniques are tremendously needed to ensure that the "implemented" RBAC is consistent with the "specified" RBAC.

Even though security policy may be embedded correctly in the application, implementation of the policy may be affected by other factors such as compilers, conversions and platforms [4]. Critical consequences arise due to existing s*ecurity vulnerabilities* in the system and those vulnerabilities may also be caused by improper reuse of software modules such as Application Programming Interface (API) [5]. In general, security vulnerabilities are the weaknesses in the system. Often proper functioning of codes being tested relies on implicit assumptions such as appropriate use of the APIs and correct reuse of the existing software. These assumptions can lead to security vulnerabilities, when the code is reused in different contexts. When these security vulnerabilities get exploited or exposed, *security violations* may occur. These violations may cause denial of service, loss of privacy, or even loss of life. Hence, these factors necessitate proper testing of the implemented security policy.

There are four kind of security testing techniques depending upon the type of security vulnerabilities or insecurities [6], [7] (classified by their cause) they intend to expose, namely *i) dependencies, ii) unanticipated user input, iii) design vulnerabilities, and iv) implementation vulnerabilities*. Insecurities caused by dependencies happen due to use of third party libraries and other interfaces. Unanticipated user input can be caused by an undesirable insertion of input. A technique to expose design vulnerabilities is an example of inserting of interfaces in the application in order to perform testing. And, implementation vulnerabilities may be useful in determining insecurities such as the man-in-the-middle attack [8] which is not considered in the application.

In the context of security, a formal verification technique verifies security policy against the security properties whereas a formal validation technique validates the design and implementation of the policy [4]. Formal validation can be performed by applying test cases which could be of two types: *positive* and *negative*. The *positive test cases* correspond to authorization states which are allowed by the access control policy and the *negative test cases* correspond to authorization states which are not allowed by the access control policy [4]. Positive test cases basically test legitimacy, whereas the negative test cases test for security vulnerabilities e.g. unauthorized access to sensitive resources.

Analyzing and managing security breaches can be expensive from various aspects. Nevertheless, security testing is inevitable. Furthermore, given the dynamic nature of RBAC access control, manual testing can be time consuming and tedious; and, may not be even sufficient. Hence, developing automatic techniques for generating positive and negative test cases is very crucial.

1.1 Approach

In this thesis, we develop a technique for creating positive test as well as negative test cases for the correct enforcement of an access control policy based on an RBAC model. One of the incentives for adopting an RBAC model is to prevent *Privilege Escalation*. This can also happen during the implementation of the code. Privilege escalation occurs when a user or an application is allowed to perform an unauthorized action. For example, a user application may access kernel level codes in an operating system and a teller may perform an unauthorized action in the bank application so that it can lead to security violations in the system.

There exists a family of RBAC models [9]. $RBAC_0$ is the most basic model and it defines “roles” to be groupings of “privileges”; “users” can be “authorized” to multiple roles and could exercise only privileges associated with authorized roles that users have activated. More complex features such as role hierarchy and constraints are part of $RBAC_1$ (Hierarchical RBAC) and $RBAC_2$ (Constrained RBAC), respectively. Role hierarchy allows roles to be organized as a *Directed Acyclic Graph* (DAG) [10] which specifies “senior” roles can inherit privileges of “junior” roles. In essence, each role is associated with its own “unique” privileges as well as those inherited from roles in the role hierarchy. The most general model $RBAC_3$ has all the aforementioned features. In this thesis, we first introduce a way to systematically test RBAC policies**.** Formally, we define the security goal for RBAC model and clarify how the positive and negative test cases help fulfill the security goal. Further, we define the semantics of positive and negative test cases with respect to role hierarchy.

In RBAC, a user can be simultaneously authorized to several roles and has the option of activating any subset of roles in a session. We espouse the principle of *Maximum Privilege* (as opposed to the principle of *Least Privilege* used in authorizing roles [1]) in order to reduce the number of testing scenarios. The rationale is that the potential to "harm" increases monotonically with the increased number of privileges. Hence, we generate tests under the assumption that each user activates all the roles simultaneously. Further, the reduction of test case is considered by avoiding generating duplicate test cases when the subDAG rooted at two authorized roles for a user overlaps. This leads to the following benefits for generating test cases: 1) positive test cases are generated only for a subset of senior-most non-dominating roles from the set of authorized roles, and 2) negative test cases are generated from the role hierarchy obtained by deleting the subDAGs rooted at these roles.

Despite the above two optimizations there could be numerous test cases generated. Further, many of these test cases may not be necessary from the security testing perspective. For instance, if it is established that a user cannot obtain privileges associated with a given role then it may be futile to generate negative tests with respect to roles that is senior to this role. With this insight, we define a new coverage criterion for negative test cases: generate test cases with respect to the (mutually non-dominating) roles which are at most *k* "fronts" from the subDAG of the role associated with a user. A *front* is a set of roles that is at the "same distance" from a given role node in the role hierarchy which is obtained by deleting the role's subDAG from the entire role hierarchy. The parameter *k* could be adjusted to control the number of negative tests to be generated (and correspondingly the level of security assurance desired).

RBAC is said to be policy neutral [3]. In other words, other security policies can be specified and enforced in conjunction with RBAC. Previously, *Multi Terminal Binary Decision Diagram* (MTBDD) based representation has been proposed to express complex security policies [11]. A policy represented in MTBDD is called PMTBDD (Policy in MTBDD). MTBDD corresponding to two different policies can be combined to obtain a PMTBDD for the combined policy. The advantages of using the PMTBDD for policy representation are its compactness and its capability in generating counter examples using theorem-proving techniques. In this thesis, we propose a Role Hierarchy MTBDD called RHMTBDD to express the RBAC's role hierarchy and show how it can be combined with a PMTBDD. Further, we show how positive and negative test cases can be generated from an RHMTBDD. Intuitively, positive and negative test cases are generated by traversing specific paths from desired (role) nodes to the appropriate terminal node in the RHMTBDD.

Building upon the security testing technique and RHMTBDD representation of RBAC, we also propose a security testing framework for RBAC policies. Our framework takes a RBAC security policy expressed in XACML – a language based on XML for specifying access control policies [12]. We use a Document Object Modeling (DOM) parser [13], [14] to extract role hierarchy information from the RBAC profile of XACML and generate the associated RHMTBDD. Further, we generate the positive and negative test cases by traversing the appropriate paths in RHMTBDD and store them in a Java `container` class.

We also evaluate our framework by generating test cases for a banking application. Our evaluation shows that both positive and negative test cases are correctly generated and that our framework can efficiently perform security testing for RBAC based systems.

1.2 Contributions

In summary, the contributions of this thesis are as follows:

- We propose a security testing approach which incorporates the principle of Maximum Privileges, structure of the role hierarchy, and a new security test coverage criterion to efficiently generate positive and negative security test cases.
- We introduce RHMTBDD that can combine various security policies to generate test cases for the more complex security policy.
- We develop a security testing framework for RBAC policies which takes a RBAC security policy expressed in XACML, parses it into a RHMTBDD representation, and then generates positive and negative test cases.
- We validate the proposed framework with in-house developed applications and the generated test cases.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. We introduce and describe background concepts and the related work in Chapter 2. The theoretical foundations for RBAC policy testing are presented in Chapter 3. Chapter 4 explains our security testing framework, each phase of the implementation, and evaluation results. Finally Chapter 5 concludes this thesis with concluding remarks and future directions.

CHAPTER 2
BACKGROUND AND RELATED WORK

In this chapter, we present foundational concepts and related work with regards to the automated security testing for RBAC we present in this thesis.

2.1 Multi-Terminal Binary Decision Diagram (MTBDD)

In this thesis, we adopt MTBDD to create security test cases to validate the implementation of the policy. MTBDD has been used in solution for many different problems. For example, it was used to create and analyze a large class of models [15]. This helped in reducing the "large state model" for a complex system to a "small scale component" and consequently harnessing state space explosion. Further, it has been also used to map Boolean vectors to integers in order to verify electric circuits [16].

MTBDD is a data structure to compactly represent a Boolean function over a set of variables [17]. An MTBDD consists of nodes, edges and terminal nodes [11]. Each node represents a predicate whether the attribute assigned to value is true or false. Each edge represents the assignment value of the predicate. And each path represents the decision label of the result of the Boolean function of the predicates in the path. MTBDD, called PMTBDD, can be used to represent a policy. Figure 1 illustrates two policies: P1 indicates that a faculty member (f) can assign grades (ag) and P3 says if a user is both a faculty member and a student, grade assignment is denied.

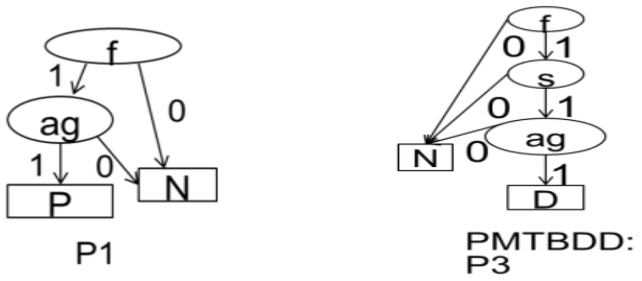


Figure 1: Policy Representation in MTBDD.

MTBDD can be used to combine different policies [11]. For example, in Figure 2, P1 allows faculty (f) to assign grades (ag), and P2 indicates student(s) can receive grades (rg). These two policies are combined to generate P3.

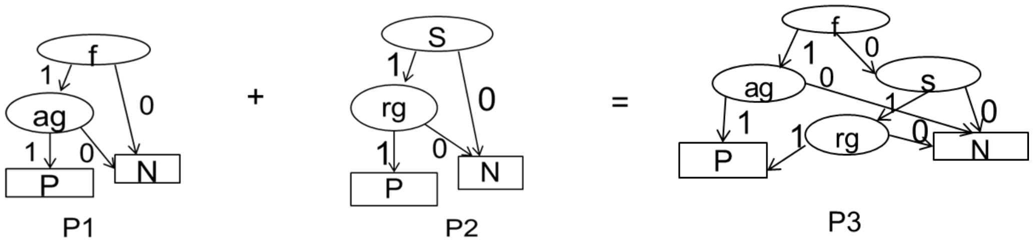


Figure 2: Combining PMTBDDs to get PMTBDD for the aggregate policy.

Access control policy can be represented with this data structure. Such representation is flexible and scalable. Further, it needs less storage space [11]. MTBDD can be suitable for finding violations and possible vulnerabilities in a policy. There are mainly three terminal nodes in the graph: *Permitted*, *Deny* and *Not applicable*. The *Permitted (P)* node terminates paths which allow the operation for a role in the policy. The *Deny (D)* node terminates paths which do not allow the operation for a role in the policy and *not applicable (N)* node terminates paths, which are not applicable for the policy.

In this thesis, we leverage MTBDD data structure and convert XACML-based RBAC policy into Role Hierarchical Multi Terminal Binary Decision Diagram (RHMTBDD).

2.2 JUnit

JUnit is a unit testing framework for Java [18] [19]. JUnit helps testers validate functionalities of source codes by using unit tests provided by the testers. A test method starts with annotation @ to differentiate from regular methods. JUnit has many features [18] [19], e.g. it offers many assertions as well as test runner for running test methods and showing the results for all test methods. Further, in JUnit testers can define a test suite which combines test classes consisting of test methods and run all the tests together.

2.3 Role-Based Access Control (RBAC)

RBAC is one of the access control mechanisms to provide the access control based on organizational structure. Each user is assigned a set of roles. Each role is associated with a set of permissions. Furthermore, each role inherits permissions based on a hierarchy [9]. Hierarchical $RBAC_1$ has following components and properties as defined in [9].

- Sets of Users (U), Roles (R), Permissions (P), and Session (S).
- $PA \subseteq P \times R$: permission to role assignment relation.
- $UA \subseteq U \times R$: user to role assignment relation.
- user: $S \rightarrow U$, a function that maps each session to a single user.
- $RH \subseteq R \times R$, role hierarchy is a partial order on R.

- roles: S $\rightarrow 2^R$, where a user can activate a session with any combination of its assigned roles and roles junior to those assigned roles.

2.4 eXtensible Access Control XML (XACML)

XACML is a general access control policy language [12]. There are many features defined in this language such as Policy, PolicySet, Rules, and Target to achieve the objective of an access control policy. This language contains PolicySet to construct policies. A PolicySet can have multiple policies or PolicySet and vice versa. A single Policy can have only one access control policy denoted by Rule. Further, this policy contains Target, which contains resources and action for the subject. XACML is used to express deny and permit actions based on the policy. XACML has a specific profile tailored for *RBAC* [20], which is used in this thesis.

RBAC profile is used for role based access control policy. Further, hierarchical RBAC model can be represented using this profile. RBAC profile contains four kinds of policies [20] such as *Role <PolicySet>*, *Permission <PolicySet>*, *Role Assignment <Policy>*, and *HasPrivilegesOfRole <Policy>* [20]. *Role <PolicySet>* determines a role attribute-value pair defined in this policy. Also, it points to the *Permission* <PolicySet> associated with a role. *HasPrivilegesOfRole <Policy>* is an option to query about the subject role in this profile. *Permission <PolicySet>* policy contains all the actions associated with a role attribute-value pair. *Role Assignment <Policy>* specifies which subjects are assigned to a role. Further, this can be used to restrict how many users (or combination of users) are allowed to activate the role. However, this policy is optional. Inheritance can be achieved in this profile by adding *<PolicySetIdReference>* and giving reference to the desired role. For example, if role$_1$ is inheriting permissions from role$_2$, it can be achieved by having a tag of *<PolicySetIdReference>* inside *Permission* <PolicySet> of the role$_1$, where *<PolicySetIdReference>* is reference to the permissions of role$_2$.

```
………………………………………………..
<PolicySet xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
PolicySetId="PPS_Manager"
PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides">
<!-- Permissions specifically for the manager role -->
<Policy
PolicyId="Permissions:specifically:for:the:manager:role"
RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
<!-- Permission to create an  account  -->
<Rule
RuleId="create a customer account"
Effect="Permit">
<Target>
<Resources>
<Resource>
<ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
<AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">Account</AttributeValue>
<ResourceAttributeDesignator
AttributeId="resource-id"
DataType="http://www.w3.org/2001/XMLSchema#string"/>
</ResourceMatch>
</Resource>
</Resources>
<Actions>
<Action>
<ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
<AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">credit</AttributeValue>
<ActionAttributeDesignator
AttributeId="action-id"
DataType="http://www.w3.org/2001/XMLSchema#string"/>
</ActionMatch>
</Action>
</Actions>
</Target>
</Rule>
</Policy>
<PolicySetIdReference>PPS_Customer</PolicySetIdReference>
</PolicySet>
……………………………………………………………
```

Figure 3 : RBAC Policy in XACML [2].

Figure 3 is the example of *Permission* <*PolicySet*> in RBAC profile. As mentioned previously, this policy specifies the permissions associated with a role. In this example, the role is Manager, referencing the role <*PolicySet*> for Manager in Line 3. Line 12 tells "Effect" is permitted if role is Manager. Further, Line 18 and Line 29 show that Manager can perform an action "credit" on resource called "Account". There is reference to permissions associated with Customer in Lines 39 with XACML feature element <*PolicySetIdReference*>. This reference tells that Manager can inherit all the actions associated with the Customer role.

2.5 Security Module/Unit Testing

Unit testing is obtained by three main actions *arrange*, *act*, and *assert* (AAA) [21]. For example, to verify the correctness of the subtract function, testers first need to act by arranging and assigning input parameters in its testing unit. Further, testers act by calling subtract function and finally assert the expected result with actual result by calling function under test. Same technique can be applied while preforming security testing. For example, in hierarchical RBAC testing, arrange is done by creating a role object and its associated permission. Further, act is done by checking whether a particular role is allowed to perform its actions. Then, the role object method is called only if it is asserted that this operation can be performed through this object.

2.6 Related Work

In this chapter, we describe how this thesis work is related to and different from other related work. Hu and Ahn [4] have done similar work to generate positive and negative test cases. Their approach involved creating negative test cases by separating *constraints* from *Access Control Model Specification (ACMF).* Access Control Model Specification consists of Model (M), Function (F) and Constraints(C) [4]. In ACMF M represents security model and F represents specification such as operation. *Constraints* represents restriction in a policy such as which role is authorized to do what operation. After separation of C (Constraints) from ACMF, constraints are given as a separate input to the formal verifier. If the formal verifier yields the result as NOT OK, a sequence of counter examples gets created and these counter examples could be used to generate negative test cases. The process of generating positive test cases is similar; however, constraints are negated when C is separated from ACMF. The formal verifier yields the result as NOT OK and the counter examples are used to generate the positive test cases. Hu and Ahn's approach involves two steps. First, it takes RBAC model and converts into RCL2000 [22]; further RCL2000 is translated into ALLOY and given as an input to Alloy Analyzer. Finally, test cases are created. In the second step, the RBAC model gets translated into Unified Modeling Language (UML) class diagram. Codes are generated through UML and test cases created in the first step are validated against the codes. Hu and Ahn's method differs from our method. Unlike their method we are not using RCL2000 and Alloy analyzer. And they use UML to generate source codes from RBAC policy. In our approach we are using RHMTBDD inspired by MTBDD to create security test cases. Further, these test cases are incorporated with JUnit specification to validate implementation of the codes.

Hu and Ahn have extended their work in [23] and designed an authorization system for a Financial Service System. They have obtained an authorization system by defining requirements, classifying RBAC objects, relations and restrictions for banking applications and finally by designing and implementing a banking application system. Further, they perform “Conformance testing” on this system which they have designed and implemented through specification. They created positive and negative test cases for this system. Positive test cases are those cases, which are allowed by the access control authorization state. Negative test cases are those cases which are disallowed by access control policy. Creation of these test cases is similar to the one described in [4] by separating constraints C from ACMF and giving as a separate input to formal verifier. To create positive test cases, they created negative constraints and gave it as an input to verifier and converted the result into counter examples which become the positive test cases. Negative cases are generated through positive constraints and results are converted into counter examples which become negative test cases. Our approach is different in generating positive and negative test cases. Their approach is based on logical model. In their approach, security testing is based on logical requirement and logical design of the system. In our approach first, we parse XACML policy to convert into RHMTBDD. Second we create positive and negative test cases by traversing the path in RHMTBDD.

Hu, Kulkarni and Ahn’s approach used Binary Decision Diagram (BDD) in order to find out anomalies in web access control policy [24]. BDD is a special case of MTBDD [11]. It has only two terminal nodes: permitted and not permitted. Their approach does not involve traversing paths to create test cases [23]. They used BDD as a set operation to find anomalies in policies.

Fisler et al. used MTBDD to verify access control property against policy defined in XACML [11]. They used MTBDD to find the impact and existence of the vulnerabilities due to the policy changes [11]. Their approach used MTBDD to determine the impact through theorem-proving. We use MTBDD to create test cases to validate the implementation of the source code against the RBAC policy, whereas their approach is focused on analyzing policies.

In summary, our approach differs from previously proposed approaches in generating test cases. Additionally, our approach utilizes MTBDD differently and modifies MTBDD to convert XACML policy into RHMTBDD with three terminal nodes “permitted”, “deny”, and “not applicable”. Our approach uses RHMTBDD to create test cases to validate the implementation of the code against RBAC model policy.

CHAPTER 3
SECURITY TESTING APPROACH

In this chapter we develop an approach for performing security testing of RBAC policy. We first start with some basic definitions followed by the detailed steps for the proposed approach.

3.1 Preliminaries

In RBAC, any privileges to perform actions on the information (objects) are controlled by grouping privileges into roles and assigning users to roles [1]. There are several models of RBAC [9]: $RBAC_0$ is the most basic; $RBAC_1$ includes all aspect of $RBAC_0$ as well as role hierarchy; $RBAC_2$ also includes all aspects of $RBAC_0$ but has no role hierarchy, but instead it has constraints on roles, privileges, and other relations (e.g. separation of duty and cardinality constraints); and $RBAC_3$ is the most comprehensive RBAC model and it inherits all aspects of both $RBAC_1$ and $RBAC_2$. In this thesis we will mainly focus on $RBAC_1$.

3.1.1 RBAC Model

There are some basic definitions associated with RBAC (from [1], for $RBAC_0$ model [9]) required for explaining our framework. The set of authorized actions (called transactions in [1], permissions or privileges in [9]) for each role r is denoted as $trans(r)$. Although [9] distinguishes between *users* and *subjects* (or *sessions*) – a user is associated with multiple subjects with several (subset) of its authorized role activated – in the following we use users and subjects interchangeably. The set of active roles of a user or a subject *s* is denoted as *actr*(*s*). The set of authorized roles of a subject *s* is denoted as *authr*(*s*). The predicate *canexecs*(*s*,*t*) denotes whether a subject *s* can perform a transaction *t* (at a given time). The functions *actr*(), *authr*(), and *canexec*() are from [1] which we use in this thesis.

Note that in the case of hierarchical RBAC ($RBAC_1$ model [9]) the set of role authorized to a subject is governed by a role inheritance hierarchy (with an associated partial order role dominance relation $\geq \subseteq R \times R$). Hence, the set of roles authorized to a subject *s* with *primary authorized role* r is all the roles *r*' appearing in the subDAG rooted at *r* in the RBAC hierarchy, i.e., $\{r' \mid r \geq r'\}$. We call non-inherited actions associated with a role as its *unique* actions.

In [1] there are three basic rules associated with RBAC. The *role assignment rule* [$canexcs(s,t) \Rightarrow actr(s) \neq \Phi$] says that a subject *s* can perform a transaction only if *s* is a member of an active role. The *role authorization* rule [$actr(s) \subseteq authr(s)$] says that for any subject *s,* only an authorized role can be activated. Note that a user can have multiple active (authorized) roles at any given time. The *rule of transaction authorization* [$canexecs(s, t) \Rightarrow t \in trans(actr(s))$] says that a subject *s* can only perform its actions associated with its active roles.

3.1.2 Goal of Security Testing

Intuitively, any violation of these three rules is associated with corresponding security vulnerabilities in the system. For example, a violation of role assignment means that a subject is able to perform an action even if it has no active roles. There are other security problems which also need to be tested for. For example, those that can be associated with availability: A user may be unable to perform an action which is in its active role set.

Hence, a **goal** of security testing should be to ensure that for every user *s* the set of all permitted actions is equal to the set of all actions associated with all active roles, i.e. , $\{t \mid canexecs(s,t)\} == trans(actr(s))$. In the following, for the sake of simplicity we define $canexecs(s) = \{t \mid canexecs(s,t)\}$.

3.1.3 Positive and Negative Test Cases

In our approach we generate two types of test cases:

1. **Positive test cases:** ensures $\mathcal{P}$: $\forall s\ canexecs(s) \supseteq trans(actr(s))$. Every subject *s* can perform all the actions corresponding to all its active roles. This is done by generating *positive* test cases to check whether a subject with a particular active role *r* can perform all unique actions associated with *r* as well as all inherited actions (in case of hierarchical RBAC) associated with *r*. The predicate $\mathcal{P}$ is ensured when all the positive test cases "pass".
2. **Negative test cases:** ensures $\mathcal{N}$: $\neg(\exists s\ canexecs(s) \supset trans(actr\ (s)))$. There does not exist a subject *s* which can perform an action that does not correspond to any of its active roles. This is done by generating *negative* test cases to check whether a user cannot perform any of its non-unique non-inherited actions. Each negative test case passes if the subject is not granted access to perform such an action. The predicate $\mathcal{N}$ is implied if all the negative test cases pass.

Combining these two sets of test cases ensures that for every subject *s*:

$$canexecs(s) \supseteq trans(actr(s)) \wedge \neg(canexecs(s) \supset trans(actr(s)))$$

$$\text{i.e., } canexecs(s) == trans(actr(s)).$$

3.2 Security Testing Approach for RBAC

In this section, we present our detailed approach for performing security testing for RBAC policy.

3.2.1 Generating tests for single authorized role

For a user with single authorized role R, the semantics of RBAC authorizes the user for all the roles in the subDAG from DAG(R) in the role hierarchy RH. The positive test cases are with regards to all the unique and inherited actions for all the roles in this subDAG. The negative test cases are generated by deleting the DAG(R) from RH. This effectively implies that actions/privileges associated with roles in DAG(R) are not inherited by any roles in RH – DAG(R). Now the negative test cases are generated with respect to the actions associated with roles in RH – DAG(R). For example if a user is authorized to a single role R, positive test cases with regard to role R are associated with permissions in the subDAG rooted at R and negative test cases are related to all the privileges associated with RH – DAG(R) as shown in Figure 4.

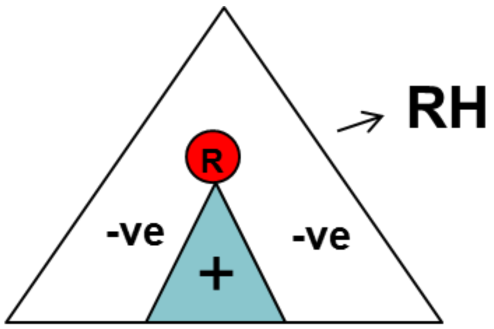


Figure 4: Positive and Negative Test Case for |auth(r)| = 1.

3.2.2 Optimizing tests for multiple authorized roles

Although in general the active role set for a user in a session is a subset of authorized roles, for the sake of simplicity, we assume that *actr*(*s*) = *authr*(*s*), i.e., all the roles a subject is authorized to ought to be activated by the subject. In essence this assumption generates test cases under *maximum privileges* assigned to a user. The rationale is that under this assumption the most serious vulnerabilities (i.e. a user can perform actions which they are never allowed) and all inconveniences (i.e. a user is not allowed to perform an action even though he is authorized to). This considerably reduces the number of test cases since there are $2^{|authr(s)|}$ different *actr*(*s*) sets possible for a given *authr*(*s*). For example, in Figure 5 a user is authorized to a role, role2. Due to RH, *authr*(*u*) = {role2, role4, role5, role6}. There are 2^4 possible subsets for this role so it creates test cases only for *actr*(*u*) = {role2, role4, role5, role6} instead of creating test cases for all different subsets.

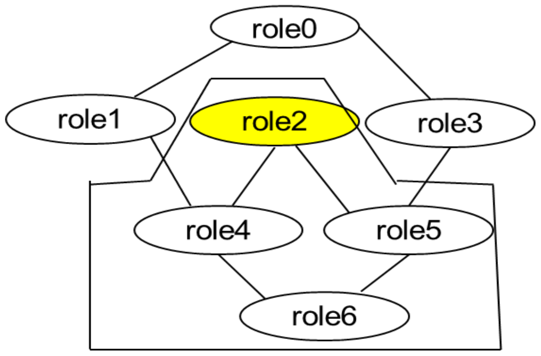


Figure 5: An example role hierarchy.

Further, assume that there exists a single most dominant role r in the *authr*(s), i.e. $r = \max_{\geq}$ (*authr*(s)), the set of positive and negative test cases can be easily generated based on all the unique and inherited actions of role r. In case there is no unique most dominant role in *authr*(s), we can determine maximal subset T of *authr*(s) such that: 1) any pair of roles in T is mutually non-dominating, i.e., if r_1, r_2 in T then neither $r_1 \geq r_2$ nor $r_2 \geq r_1$ unless $r_1 = r_2$; and 2) If r_1 in T then there is no r_2 in *authr*(s) such that $r_2 \geq r_1$ unless $r_1 = r_2$. Basically, T consists of senior-most mutually non-dominating roles from *authr*(s). The positive and negative test cases can now be computed by taking the union of all the unique and inherited actions of roles in T. This procedure avoids generating duplicate test cases since any role junior to multiple roles in T is only considered once.

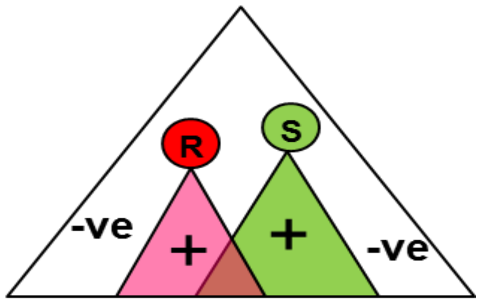


Figure 6: Optimizing test cases for mutual non-dominating role.

Consider roles R and S in Figure 6. They both are mutually non-dominating roles. Optimization in creating test cases can be achieved by not duplicating the test cases for overlapping area. Suppose in Figure 5 *authr*(*s*) = {role4, role5, role6}. We can calculate maximal subset *T* = {role4, role5}. Positive test cases can be generated by taking union of all unique and inherited actions of role4 and role5 (that is the union action of role4, role5 and role6). And negative test cases can be generated for which are not included in the union set of positive tests. Positive test cases = {union actions: inherited actions of role4} ∪ {union actions: inherited actions of role5}. Negative test cases are ⌐{{union actions: inherited actions of role4} ∪ {union actions: inherited actions of role5}}.

3.2.3 Optimizing Negative Test Cases

This testing coverage criterion is to test for all vulnerabilities related to actions for each role. It creates positive test cases for every role's unique action and inherited actions. And negative test cases are created based on all actions which are not inherited and unique actions for a role. However, despite the optimizations suggested in the previous chapters the number of test cases could be enormous.

In this section, we first provide a motivating example and then describe a new test coverage criterion which can be used to balance the number test cases and the desired level of assurance.

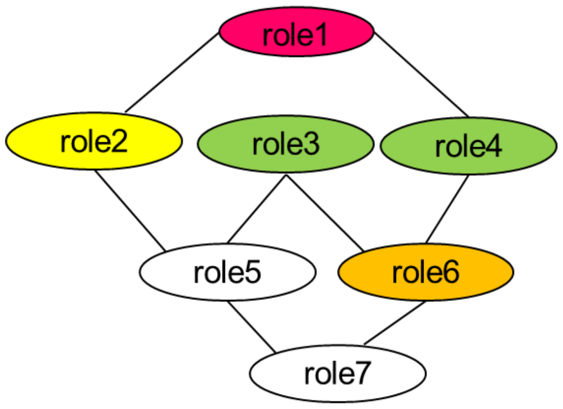


Figure 7: Reducing Negative Test Cases.

We can optimize negative test cases by considering only min under dominance role of non-authorized role r's unique action such that $r = \min_{\geq} (\mathit{RH\text{-}authr}(s))$, where *RH* is the role hierarchy. Negative test cases can be easily generated based on all the unique and inherited actions of role *r*. Consider if user is authorized to role2 in Figure 7. Negative test case for role1 relates to all the actions of min $_{\geq}$ [{role1, role2 role3, role4, role5, role6, role7} – {role2, role5, Role7}] = min $_{\geq}${role1, role3, role4, role6} = role6. We can notice that this reduces the number of negative test cases by eliminating actions of role1, role3, and role4. This is under the assumption that if a user is unable to perform actions associated with a role *r* then the user would unlikely be able to perform actions associated with a role senior to *r*. In essence, we assume that the potential to harm monotonically increases with the number of privileges (active roles).

3.2.3.1 Test coverage criteria

Motivated by the above example, we suggest the following coverage criteria: generate test cases only with respect to the (mutually non-dominating) roles which are in at most *k* "fronts" from the subDAG of the role associated for a user. A *front* is a set of roles which are the "same distance" from a given role node in the role hierarchy obtained from deleting the role's subDAG from the entire role hierarchy. The parameter *k* could be adjusted to control the number of negative tests to be generated (and correspondingly the level of security assurance desired). Figure 8 illustrates the optimization of creating negative test cases.

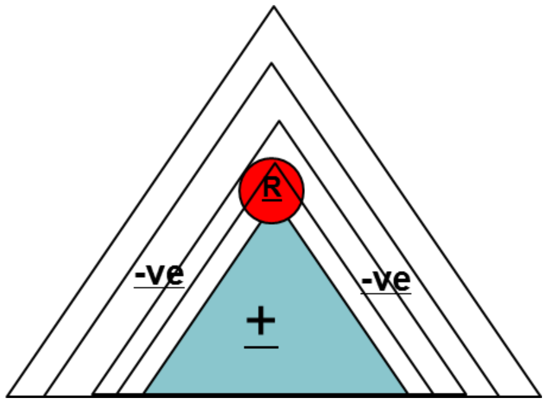


Figure 8 : Fronts Coverage Criteria.

3.3 Role Hierarchy MTBDD (RHMTBDD)

Figure 9 (a) is an example of a hierarchical RBAC model. In this policy, a manager (a user with role Manager) can create and cancel the account, a customer (a user with role Customer) can transfer and change the account, and a teller (a user with role Teller) can deposit and withdraw the account. Furthermore, manager can inherit all the permissions from customer and teller. Customer can also inherit all the permissions from Teller. Figure 9 (b) is another example of hierarchical RBAC model, where Manager is inheriting from two same level junior role Customer and Agent. Further, Customer and Agent cannot inherit each other unique actions. Table 1 (top) and (bottom) has the unique actions and inherited actions assigned to corresponding roles illustrated in Figure 9 (a) and Figure 9 (b), respectively.

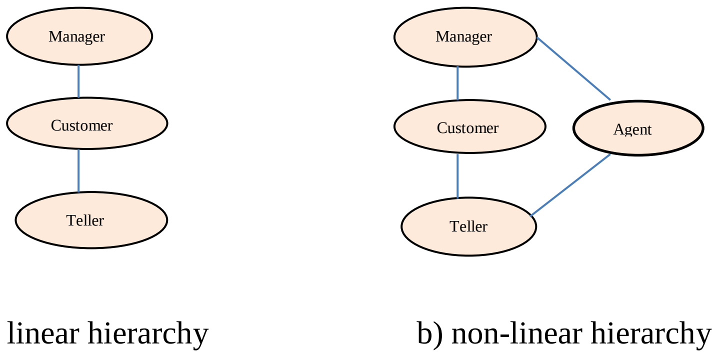


Figure 9: Hierarchical RBAC examples.

Table 1: Unique and Inherited Action for Figure 9a) (top) and Figure 9b) (bottom).

Role	Unique Action	Inherited Action
Manager	Credit, Cancel	Transfer, Charge, Suspend
Customer	Transfer, Charge	Deposit, Withdraw
Teller	Deposit, Withdraw	None

Role	Unique Action	Inherited Action
Manager	Credit, Cancel	Transfer, Change, Suspend
Customer	Transfer, Check	Deposit, Withdraw
Teller	Deposit, Withdraw	None
Agent	Suspend	Deposit, withdraw

We construct RHMTBDD with *role nodes, action nodes and decision nodes* as well as three terminal nodes *permitted* (*P*), *denied* (*D*) and *not applicable* (*N*) (described later in Chapter 4. Note not every RHMTBDD may have all the three terminal nodes.). Each role node represents a predicate corresponding to a role attribute-value pair such as "role = Manager". Similarly, each action node represents predicate corresponding to an action attribute-value pair such as "action = credit". We introduce a new node called the decision node in order to distinguish between the two same level inheritances (if they exist) for any role such as shown in Figure 9 (b). As defined in [11] MTBDD has three useful characteristics. We have adopted these characteristics to build RHMTBDD. First, a MTBDD enforces a fixed ordering among the various nodes and all “policy” paths are traversed from a root node (there can be multiple root nodes corresponding to senior most non-dominating roles in the role hierarchy) to a terminal node. Figure 10 and Figure 11 are examples of RHMTBDD. The paths 101 (denoting *a Manager can perform the cancel operation*) and 011 (denoting *a Customer can perform transfers*) are examples of policy paths starting from the senior most Manager role node in Figure 10. Restricted or fixed ordering among the nodes enables MTBDD to be a canonical representation of the policy. Second, in MTBDD any sub tree can appear only once as is the case in Figure 10 and Figure 11. This property makes MTBDD a compact representation of the policy. Third, MTBDD deletes irrelevant nodes from the MTBDD. For example, in Figure 10, we do not have *D* terminal node since there are no paths lead to it.

In our sample RHMTBDD, an ellipse shape corresponds to a role node, a rectangle with round corner corresponds to action nodes and a rectangle with sharp corner corresponds to terminal nodes. Figure 10 is a graphical representation of the policy. In this figure Manager (M), Customer (C), and Teller (T) are role nodes. Credit (cr), cancel (cn), transfer (tr), charge (ch), deposit (d), and withdraw (w) are action nodes. Further, a rectangle with sharp corner corresponds to the end of the path called terminal nodes and denotes as permitted (P) and not applicable (N). Each node has two outgoing edges. Left edge is labeled as 1 and right edge is labeled as 0 in each RHMTBDD. If the left edge of the node is taken the attribute-value of the node corresponds to *true* otherwise, if the right edge is taken then it corresponds to *false*.

If we take the path starting from the root 0011 (Teller can deposit), 100001 (Manager can deposit), and 1001 (Manager can transfer), we can see these paths correspond to valid actions. The process for creating positive test case is to follow paths from a desired node and follow until it has reached the permitted node. Previously mentioned paths 0011, 100001, and 1001 are all positive test cases because they lead to the permitted terminal node.

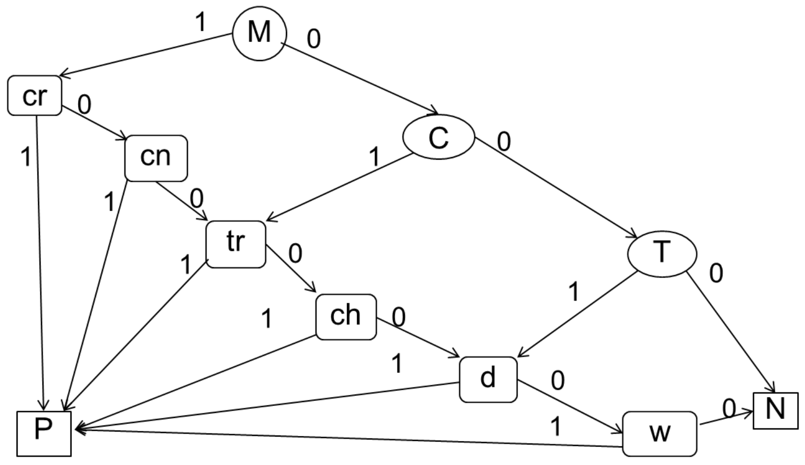


Figure 10: RHMTBDD for Figure 9 (a).

3.3.1 Introduction of The Decision Node

We introduce a new node called *Decision Node* (DN) to create RHMTBDD for $RBAC_1$ model. The objective of DN is to assist RHMTBDD, if there are two same level inheritances for a senior role. DN is created when a role object inherits from two different junior roles at the same level. DN is added as the *left child node* of this role. And the first junior role is added as the right child of this role node and the second junior role is added as the right child node of the first junior role node.

Manager inherits two different role objects *Customer* and *Agent* at the same level in Figure 9 (b). DN is added to the left of the *Manager* node and the *Customer* node is added as the right child of the *Manager* node. Further the *Agent* node is added as the right child of the *Customer* node in the RHMTBDD. Figure 11 depicts how DN helps to determine if there are two same level inheritances (mutually non dominating roles) and how DN restricts to perform actions among mutually non dominating roles. In Figure 11 there is no path from *Agent* to *Customer* action and vice versa. This implies that *Agent* cannot perform unique actions associated with *Customer* and *Customer* cannot perform *Agent's* action as stated in Figure 9 (b). *Manager* can perform all the actions as it inherits from all the roles. We apply this algorithm with more complex hierarchical RBAC model, where DN helps disallow any role to perform its unauthorized action.

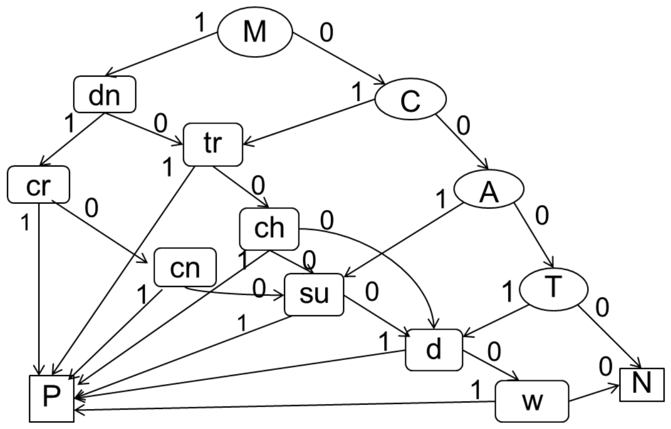


Figure 11: RHMTBDD for Figure 9 (b).

3.3.2 Structure of RHMTBDD

This section describes the layout of RHMTBDD. The RHMTBDD is defined by the following three rules:

1. **Role with no inheritances**: In this rule, first unique action node of this role node is the left child for this role node; and, each subsequent unique action node is the right child of the previous unique action node. Each unique action node's left child is the terminal node P. Further, last unique action's right child is the terminal node N. The right child of this role node is another role node or the terminal node N.
2. **Role with single inheritances**: When a role in the inheritance hierarchy has a single inheritance (i.e. the role node inherits from a single junior role), the node for the first unique action of this role is its left child. Further, the right child of this role node is its junior role node; and, each subsequent unique action node is the right child of the previous unique action node. And finally, the last unique action's right child is the left child of its junior role.
3. **Role with double inheritances**: In this case, when there are two junior role inheritances; the left child of this role node is a DN. The right child of this role node is its first junior role node. Further, its first unique action is the left child of the DN; and, each subsequent unique action node is the right child of the previous unique action node. Further, each unique action's left child is the permitted node. The DN's right child is the left child of the first junior role

node; and, DN’s left child is its first unique action. Further, the last unique action’s right child is the left child of the second junior role.

CHAPTER 4
SECURITY TESTING FRAMEWORK: IMPLEMENTATION DETAILS AND EVALUATION

4.1 Security Testing Framework

Our framework takes as input an RBAC policy specified in XACML. The Policy Parser converts the RBAC policy into a policy graph represented in RHMTBDD. Using RHMTBDD, it generates test cases by following appropriate paths in the RHMTBDD from the root node to a terminal node, creating respectively the positive and negative test cases for allowed and disallowed actions in the policy. These test cases can be used in testing tool such as JUnit for testing the system based on the specified RBAC policy. Our security testing framework is illustrated in Figure 12 .

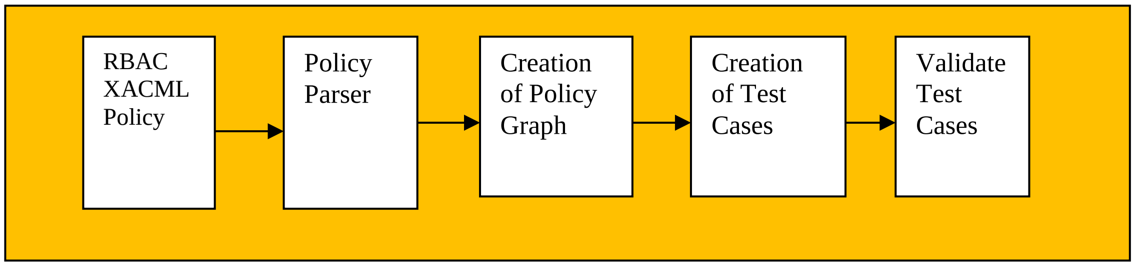


Figure 12: Implementation framework for creating Test Cases.

4.1.1 XACML Policy Parser

The Policy Parser parses the *Permission<PolicySet>* associated with each role in the RBAC profile of the input XACML file. The parser uses DOM APIs [13], [14] to accomplish this task. The parser basically keeps track of inheritance hierarchy and maintains a list of the unique actions and inherited actions for each role.

4.1.1.1 Implementation of parsing policy

We used Java DOM API to parse policies written in XACML [14]. DOM stores XML file as a tree structure. This Parser sets the root node and returns all children associated with this node. In order to access all the children nodes and their attributes, DOM provides many APIs depending on the need of parsing [14]. Further, this parser extracts information related to every role, inheritance, and their action and stores such information in a linked list. Furthermore, we used this information to create RHMTBDD corresponding to the policy written in XML. Figure 13 is an excerpt of the Policy Parser code. It finds "`PolicyId`" associated with "`Policy`" for *role* and stores this information in new object called *`RoleCreator`*. The *`RoleCreator`* Class stores the information of each role, its permission, and its inheritance.

```
NodeList nList = doc.getElementsByTagName("PolicySet");
for (int temp = 0; temp < nList.getLength(); temp++){
Node nNode = nList.item(temp);
NodeList children = nNode.getChildNodes();
for (int j = 0; j < children.getLength(); j++) {
    Node child = children.item(j);
    String childName = child.getNodeName();
    RoleCreator rolecreator;
    if (childName.equals("Policy")) {
        Element eElement = (Element) child;
        rolecreator = new RoleCreator();
        String role = eElement.getAttribute("PolicyId");
        rolecreator.setRoleName(role);
                    ..........................................
```

Figure 13 : Parsing XACML Policy.

Figure 14 has the output through parsing RBAC policy written in XACML. Lines 2, 7, 11, and 14 have role content such as Manager, Employee, Teller, and Agent, respectively. Further, lines 3, 8, 12 and 15 have RuleId as to what action they can perform. Lines 4 and 5 display that Manager can inherit from Employee as well as Teller. Further, Lines 17 displays roles, permission and inheritance described in policy.

```
1.  Permissions:specifically:for:the:manager:role
2.          manager
3.          RuleId="sign a purchase order"
4.          ChildContentPPS_Employee
5.          ChildContentPPS_Teller
6.  Permissions:specifically:for:the:employee:role
7.          employee
8.          RuleId="create a purchase order"
9.          ChildContentPPS_Teller
10. Permissions:specifically:for:the:teller:role
11.         teller
12.         RuleId="look at purchase order"
13. Permissions:specifically:for:the:agent:role
14.         agent
15.         RuleId="update a purchase order"
16.         ChildContentPPS_Teller
17. [|manager: [sign], [PPS_Employee, PPS_Teller]|, |employee: [create], [PPS_Teller]|, |teller: [look],
    [none]|, |agent: [update], [PPS_Teller]|
```

Figure 14 : Output of XACML Policy Parsing.

4.1.1.2 Implementation of RHMTBDD

This section describes the algorithm called DAGCreator to generate RHMTBDD corresponding to the parsed RBAC policy. DAGCreator starts with the roles in reverse topological order in the role hierarchy. This ensures that RHMTBDD is already constructed for all the junior roles when a senior role is processed. It checks if the current role node has one or two inheritances at the same level. The algorithm can be generalized if the role inherits from more than two junior roles at the same level – a role hierarchy in which a node inheriting from more than two junior roles can be converted into a hierarchy with each node having only two juniors by introducing some "virtual" or dummy roles. A similar technique of introducing additional roles is used to prevent inheritance of certain “private” privileges in the role hierarchy [25]. The structure of RHMTBDD for a role hierarchy in which there are at most two junior roles for any role node has been described in the previous chapter. Recall that, if the role inherits from just a single junior role, then the first unique action as an *action node* – a node which denotes a privilege - is added as the left child of the role node. Note that the *edge label* for the edge to the left child is always 1 – denoting “true” - and that to the right child is 0 – denoting “false” value for the predicate corresponding to the node from which the edge originates. Subsequently each unique action of this role node is added as the right child of each of the subsequent action node. Each action node’s left child is the *permitted node* (*P*) since these actions are permitted by the policy. The role node corresponding to the junior role becomes the right child of the role node. Additionally, the first unique action of the junior role node is added as the right child of the last unique action of the role node. The action nodes corresponding to remaining unique actions of the junior role are added in a similar manner to those for the role node. Further, the least privileged node ends with *not applicable* (*N*) node. Figure 9 (a) is an example of this inheritance type.

If a role inherits from two junior roles, then the algorithm adds one *Decision Node* (DN) as the left child of the role node. Further, it adds the first junior role node as the right child and the second junior role node as the right child of the first junior role node. The action node corresponding to the first unique action of this role node is added as the left child of DN. The first unique action node of the first junior role becomes the right child of the DN. And the first unique action node of the second junior role becomes the right child of the last unique action of this role node. The rest of the rule remains the same as the one in a single level inheritance. Figure 9 (b) has this kind of role hierarchy and their corresponding RHMTBDD in Figure 11.

4.1.2 Generation of Test Cases from RHMTBDD

The test cases can be easily generated by traversing paths over the RHMTBDD as follows:

- **Generating positive test case for a given (single authorized) role:** it simply performs *depth-first search* (DFS) [10] on the left subDAG of the role node. A positive test case is generated for each action node which can reach the permit terminal node.
- **Generating positive test case for a multiple authorized role:** We consider **state-preserving** (links that have been already visited) invocation of a sequence of DFS, subsequently starting from role nodes in the authorized set. The links visit while each DFS are marked and preserved for subsequent invocation of DFS. Each invocation of DFS traverses only those links which have not yet been visited by any previous DFS invocation. Positive test cases are generated by the steps of the previous case (i.e., whenever an action node can reach the permit terminal node). The state preserving invocation ensures that the duplicated test cases are not generated (i.e., conceptually it performs the union of subDAGs rooted at the role nodes in the authorized set (see Section 3.2.2)).
- **Generating negative test case for single (multiple) authorized role:** We again use the state preserving invocation of DFS. In this case, the DFS starts with RHMTBDD with the link state of all the links visited during generation of

positive test cases preserved. Further, the permitted P terminal node is replaced by denied D. The negative test cases is now generated by performing DFS for the role nodes which were not covered during the positive test case generation which is similar to the way for generating positive test cases – with a simple difference that the paths are traversed to the D terminal and consequently test-cases now are negative test cases. Intuitively, this process simply deletes the subDAGs corresponding to authorized roles from the overall RHMTBDD and generates (negative) test cases on the remaining RHMTBDD.

- The above process for generating negative test cases can be refined to generate test cases only up to certain *fronts* given the role nodes in the fronts which can be obtained from the RH. In this case, it only generates the test cases for role nodes in the fronts following the above procedure started with the marked RHMTBDD obtained after generating all the positive test cases. The computation of the front on the role hierarchy is as follows. First, it removes the subDAGs associated with the roles in the authorized set from RH. The first front is the set of leaf node in the remaining DAG. The second front is the set of leaf role nodes, which in the DAG is obtained by deleting the first front and so on.

Since the complexity of performing DFS is O(n+m), where n is the number of nodes and m is the number of edges in the graph, the complexity for generating all the test cases is same, where n is the number of nodes and m is the number of edges in the RHMTBDD. This can be established by the simple observation that the execution of all the (mini) DFSes simply amounts to the execution of one (large) DFS over the entire RHMTBDD due to the state-preserving nature of the invocations.

4.1.3 Approaches for Creating Positive Test Cases

There are two approaches for creating positive test cases. The first approach is used when a desired role and desired action are given and it finds the result by traversing RHMTBDD till it reaches a terminal node which would be either "*permitted*" node or "*not permitted*" node. Table 2 shows the first approach for generating positive test cases for `CheckOperation(Role, Roles' authorized action)`. This method finds the role's authorized action and checks whether the path ends with the terminal node "*permitted*". The second approach is to generate all the positive tests associated with a given role. Table 2 lists positive test cases for `CheckAllOperations(Role)` to find role and associated results for all unique and inherited actions by traversing RHMTBDD. For example, in Table 2 the positive test cases will return "true" because they lead to a permitted node in RHMTBDD shown in Figure 10.

Table 2: Example of Positive Test Cases.

Approach 1	**Approach 2**
CheckOperation(Role, Role's authorized actions)	**CheckAllOperations(Role)**
1. CheckOperation(Manger, Credit)	1. CheckAllOperations(Manager)
2. CheckOperation(Customer, Transfer)	2. CheckAllOperations(Customer)
3.CheckOperation(Manager, Cancel)	3. CheckAllOperations(Teller)
4.CheckOperation(Customer, Withdraw)	

Table 3: Example of Negative Test Cases.

checkOPerationNegative(Role, Roles non unique action and non-inherited action)	**checkOPerationNegative(Role)**
1. CheckOPerationNegative(Teller, credit)	1. CheckOPerationNegative (Manger)
2. CheckOPerationNegative (Teller, cancel)	2. CheckOPerationNegative (Customer)
3. CheckOPerationNegative (Teller, transfer)	3. CheckOPerationNegative (Teller)

4.1.4 Approach for Creating Negative Test Cases

Creating negative test case is to find all actions which are not unique and inherited action for any role in RHMTBDD and generate test cases for those actions. This approach enables to test all the vulnerabilities for all those actions on which a role should not perform. Intuitively, we need to generate a list for unique actions and inherited actions for each role. Further, we create a function in which we take out all the unique and inherited actions out of this list and create negative test cases with these actions. Table 3 is an example of negative test cases for RHMTBDD in Figure 10. As mentioned earlier state preserving DFS is used for this purpose.

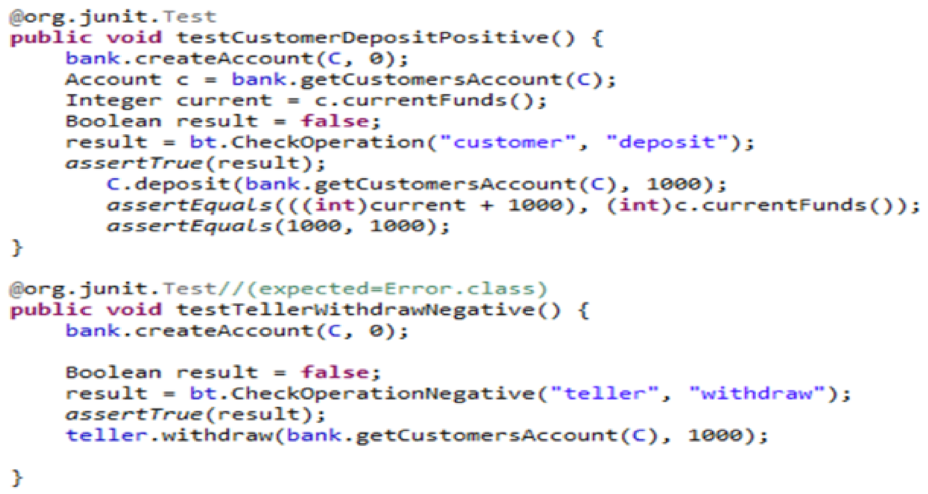
4.2 Summary of Framework Implementation

In summary, we keep all the information such as *unique actions* (for a role) and its *inherited list* (the list of roles junior to this role) during the parsing in an object called `RoleCreator`. The Parser object returns a list of `RoleCreator` objects corresponding to all the roles in the role hierarchy. We give this information to the class called `DAGCreator.` This class looks at each role's unique actions and inheritances and creates the RHMTBDD. Further, we have created a class called `container`, where we store all security test cases from RHMTBDD for all the roles.

4.3 Execution of security test cases

Test cases generated from RHMTBDD are validated in JUnit by developing a small banking application to find security violations. The banking application contains Account, Bank, Manager, Customer, and Teller class. Account class contains methods for querying information such as who is the owner of the account and for operations such as withdraw fund, deposit fund and check fund for the account. Further, Bank class contains information about manager, a list of customers and a list of tellers. Hierarchy of this role is as illustrated Figure 9 (a). In this policy, Manager can perform “credit” and “cancel” actions. Customer can perform “deposit” and “withdraw” action. And Teller can perform unique actions “check”.

Since JUnit does not have access control mechanism, we use the same mechanism for generating the test cases in simulating the access control during run-time, i.e., traversing an RHMTBDD to find whether an action is permitted or not for a given role. Note that the RHMTBDD used for generating the test case is different from RHMTBDD used during testing, since RHMTBDD corresponds to the policy that is actually implemented by the system. In summary, we used the following step for validating our testing framework: 1) We used the correct RHMTTBDD to generate the test cases as well as to simulate the implemented access control – in this case all tests, both positive and negative, were passed; and 2) We used the correct RHMTBDD to generate the test cases and incorrect RHMTBDDs for simulation -- in this case we verified whether the test cases which needed to be failed or passed return the valid results.



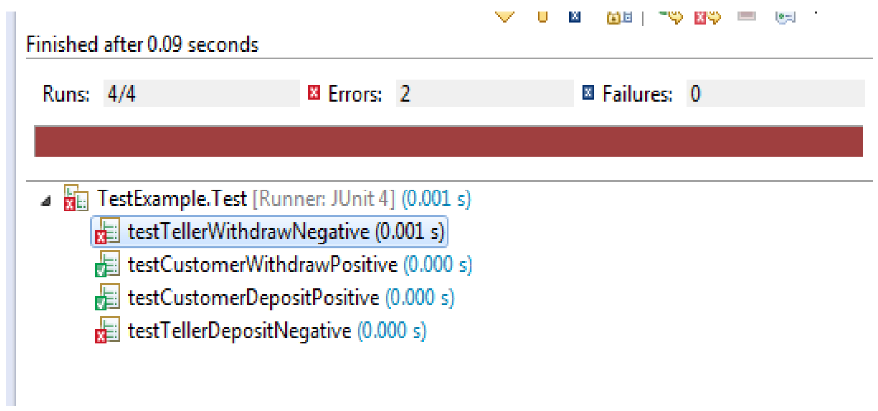
```
@org.junit.Test
public void testCustomerDepositPositive() {
    bank.createAccount(C, 0);
    Account c = bank.getCustomersAccount(C);
    Integer current = c.currentFunds();
    Boolean result = false;
    result = bt.CheckOperation("customer", "deposit");
    assertTrue(result);
       C.deposit(bank.getCustomersAccount(C), 1000);
       assertEquals(((int)current + 1000), (int)c.currentFunds());
       assertEquals(1000, 1000);
}

@org.junit.Test//(expected=Error.class)
public void testTellerWithdrawNegative() {
    bank.createAccount(C, 0);

    Boolean result = false;
    result = bt.CheckOperationNegative("teller", "withdraw");
    assertTrue(result);
    teller.withdraw(bank.getCustomersAccount(C), 1000);

}
```

Figure 15: Security Test Cases.

Figure 15 displays an excerpt of security test cases generated for the application. The first test case `testCustomerDeposit()` is a positive test case. In it a customer account is created and `bt.checkOperation` is called. The object `bt` is an RHMTBB object which represents the implemented RBAC policy. The method `bt.checkOperation` basically traverses the RHMTBDD to determine whether according to implemented policy this action is permitted or not – it returns true when the operation is permitted. Then deposit action is asserted by calling `deposit` method. Further, the test asserts whether the amount has been credited to the customer account by checking the current fund in the account. The last part is actually not part of security testing but functional testing. We have considered this portion here to simply show how security and functional testing can be performed together.

The second test `testTellerWithdrawNegative()` is an example of negative test case. We follow the same steps for asserting negative test cases as we go through for the positive test cases. The difference is that a different object, `bt.checkNegativeOperation` is called which returns true when the operation should fail according to the RHMTBDD of `bt`.

Figure 16 shows an output of running all the test cases. For example, "`testCustomerWithdrawPositive`" and "`testCustomerDepositPositive`" are the results of positive test cases. And "`testTellerWithdrawNegative`" and "`testTellerDepositNegative`" are the results of negative test cases. As we can notice, there exist two errors. These show if the policy is implemented correctly and which operation failed as well.



```
Finished after 0.09 seconds

Runs: 4/4        Errors: 2        Failures: 0

TestExample.Test [Runner: JUnit 4] (0.001 s)
    testTellerWithdrawNegative (0.001 s)
    testCustomerWithdrawPositive (0.000 s)
    testCustomerDepositPositive (0.000 s)
    testTellerDepositNegative (0.000 s)
```

Figure 16: Results of Security Test Cases.

4.4 Evaluation

4.4.1 Evaluation of Maximum Privileges

In this section, we evaluate our strategy for reducing the number of test cases. Recall that we generate tests under **maximum privilege** *actr(s) = authr(s).* Additionally, it exponentially reduces the number of test cases. In order to quantify the advantage of this strategy we study the following three scenarios:

1. **Flat scenario**, where role hierachy is not considered. Each role is considered to be independent role.
2. **RH scenario**, where RH is considered. That is, if a user has activated a role, it means it has actived all the role in the subDAG rooted at the very node in the RH.
3. **Maximum privilege scenario**: in this case a user is assumed to have activated the maximum privileges pertained to her roles in accodance to the role hiearchy.

For our evaluation we consider the example role hierarchy shown in Figure 9 (b) where there are four roles: Manager, Customer, Agent, and Teller. Assume that the Manager role (r_m) has one unique permission; Customer (r_c) and Agent (r_a) roles have two unique permissions; and Teller (r_t) role has one unique permission. In the following we analyze the number of positive and negative test cases alsong with the total number of test cases that will be generated for the above-mentioned three scenarios when the user is authorized to various roles in the hierarchy.

First, consider a user is authorized to all the four roles under the flat RBAC scenario. There are 2^4 active sets. For each active, set we calculate the unique permission over all the active roles in that set. For example, if an active set has only Manager and Agent roles then the number of positive test cases would be 3 since the Manager is authorized to one permission and the Agent is authorized to two permissions. Further, the number of negative test cases will also be 3, since the number of permissions over the two other roles, Customer and Teller, sums to three. We carried out similar computation for all the subsets and obtained the number of positive and negative test cases.

In the second senario, we considered RH. In this case there are only 6 distinct active set {},{r_m}, {r_c}, {r_a}, {r_c, r_a}, {r_t}. The reason is all other active set are equivalent to one of these active sets. For example if we take {r_m,r_a}, this simply equals to {r_m} because Manager inherits Agent. We calculated (number of positive, number of negative) test cases for these six representative active sets and we get the following results, repectivley: (0,6), (6,0), (3,3), (3,3), (5,1), (1,5), producing the total of 18 positive and 18 negative test cases.

In the third scenario, we considered RH + MP. In this case, there is only one active set which is {r_m} where all the roles are active since manager inherits all the roles. We have 6 positive and 0 negative test cases.

Figure 17 that RH+MP substantially reduces the number of test cases. For example, for the total number of security test cases, the case for RH+MP has only 24 test cases whereas RH scenario generated almost 78 test cases and the flat scenario had over 154 test cases.

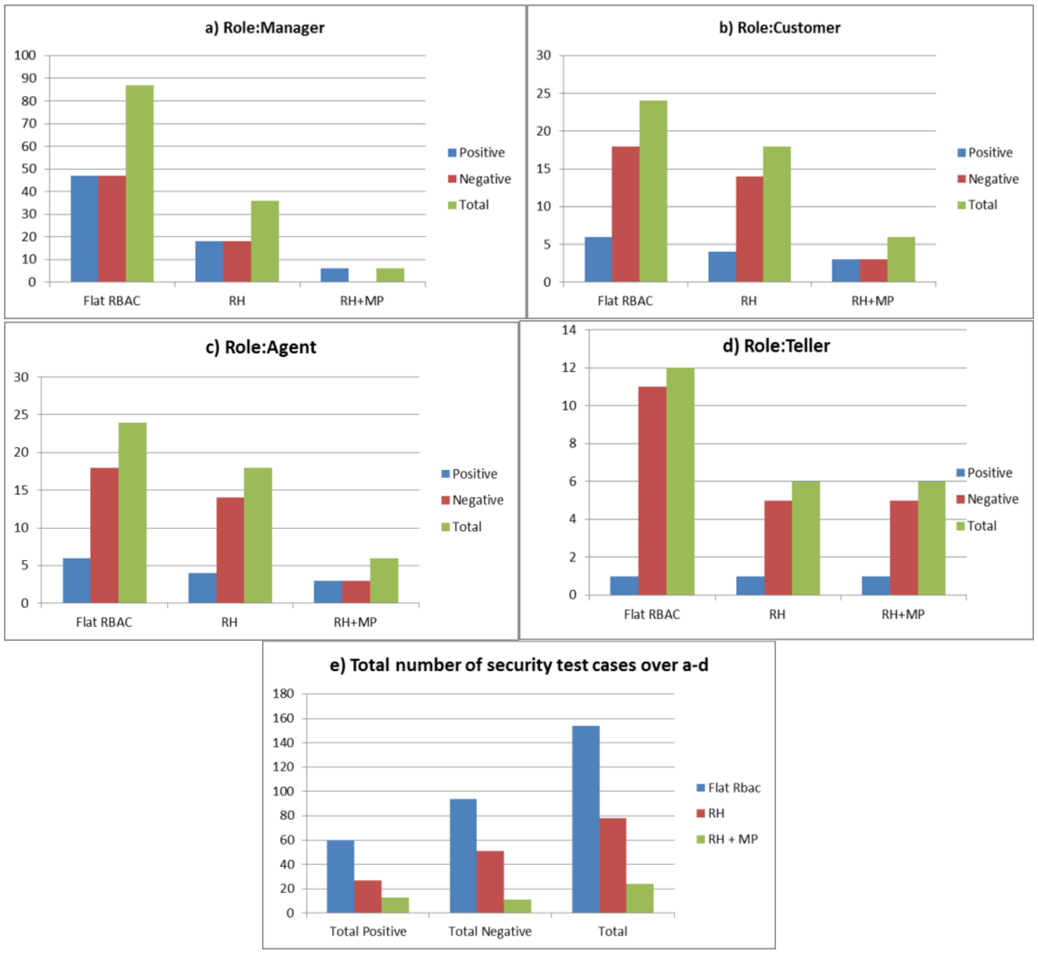
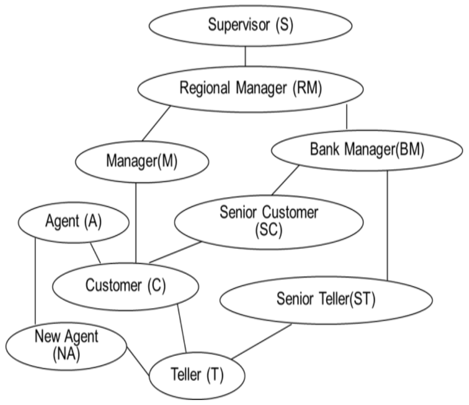


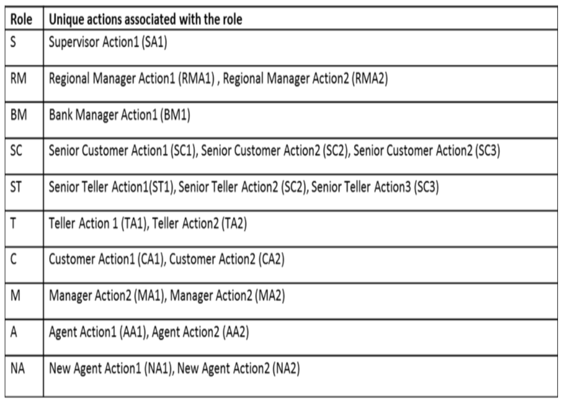
Figure 17: Evaluation Results.

4.4.2 Evaluation of Fronts Coverage Approach

In this chapter we evaluate the effectiveness of front coverage approach. Consider the role hierarchy in Figure 18 (a). The table in Figure 18 (b) gives the unique actions corresponding to each role in the role hierarchy.



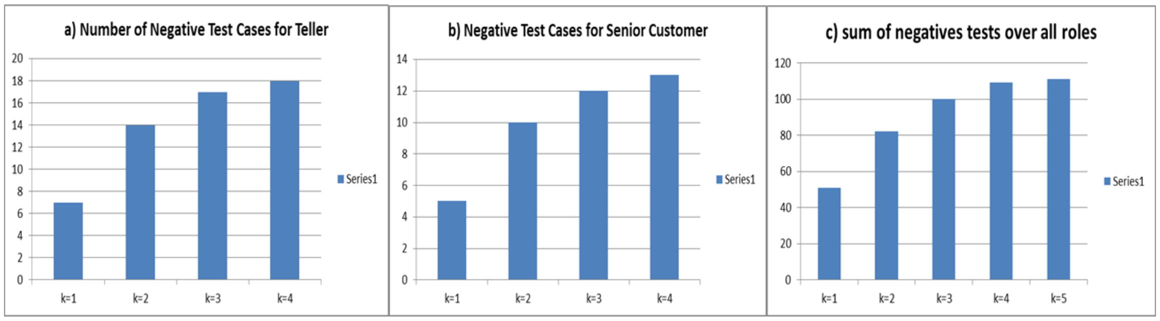
a)



Role	Unique actions associated with the role
S	Supervisor Action1 (SA1)
RM	Regional Manager Action1 (RMA1) , Regional Manager Action2 (RMA2)
BM	Bank Manager Action1 (BM1)
SC	Senior Customer Action1 (SC1), Senior Customer Action2 (SC2), Senior Customer Action2 (SC3)
ST	Senior Teller Action1(ST1), Senior Teller Action2 (SC2), Senior Teller Action3 (SC3)
T	Teller Action 1 (TA1), Teller Action2 (TA2)
C	Customer Action1 (CA1), Customer Action2 (CA2)
M	Manager Action2 (MA1), Manager Action2 (MA2)
A	Agent Action1 (AA1), Agent Action2 (AA2)
NA	New Agent Action1 (NA1), New Agent Action2 (NA2)

b)

Figure 18: Example of a) role hierarchy for front coverage and b) the corresponding unique actions associated with each role in the hierarchy.



a) b) c)

Figure 19: Results of FCC for a) Teller role and b) Customer role; and c) Over all the roles for various k values.

We compute the number of negative test cases that will be generated for various roles in the hierarchy while the value of *k* (the number of fronts that are considered) varies. Figure 19 (a) shows the number of negative test cases generated for *k*=1 through 5 for the Teller role. Similarly, Figure 19 (b) shows the number of negative test cases generated for *k*=1 through 5 for the Customer role. Finally, Figure 19 (c) shows the number of negative test cases generated for *k*=1 through 5 over all the roles in the role hierarchy in Figure 18. From this example, we could observe that the substantial reduction in the number of test cases was obtained for a lower *k* value. For example, for *k* =1, almost 54% reduction was occurred while 26% reduction was observed for *k*=2. Obviously this reduction may vary with the structure of the role hierarchy as well as the distribution of unique privileges over the roles in the hierarchy. However, our evaluations clearly demonstrate the feasibility and practicality of our approach.

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSIONS

In this thesis, we have developed a technique for testing RBAC policies. It is critical to test RBAC policies since each policy is essential to meet and enforce the security requirements for preventing and detecting any security violations in the context of the access control requirements. Our technique incorporates the principle of maximum privileges, the structure of role hierarchy and a new security test coverage criterion to efficiently generate positive and negative security test cases. We developed RHMTBDD which is an extension of MTBDD to represent the RBAC policy. Further, we designed and implemented security framework to validate RBAC policies. This framework first parses the policy written in XACML to create RHMTBDD and then generates test cases by traversing the RHMTBDD. These test cases are created and stored in a container class. We validated and evaluated our security testing framework using JUnit as well as evaluated the saving obtained by our security testing technique through case studies.

5.2 Future Work

There are several future research directions for our work presented in this thesis. We list a few of potential future research tasks.

5.2.1 Static Separation of Duty (SoD)

One of the main reasons for popularity of RBAC is that it is easy to specify separation of duty constraints. Such constraints are part of $RBAC_2$ (as well as $RBAC_3$). (Static) SoD constraints specify mutually exclusive roles and prevent security vulnerabilities which may be caused by a user having multiple roles which may allow him to commit fraud. For example in a bank application, the Billing Clerk role and the Account Receivable role should not be assigned to the same user. Hence, these roles can be declared to be mutually exclusive by a SoD constraint. Note that with respect to the given role hierarchy, if r1 and r2 are mutually exclusive roles then r1 cannot be a senior (containing) role to r2 or vice versa.

Clearly, checking for violation of SoD constraints is part of the negative testing. For example, negative test cases for a user who is assigned to Billing Clerk role, will include a test where it would check whether the user can perform unique actions of the Account Receivable role. Hence, SoD constraints can be easily considered and these constraints can become part of negative test cases.

RHMTBDD can be combined with SoD PMTBDD. For example, in Figure 20 policy P1+ P2 PMTBDD and P3 PMTBDD are combined to obtain (rightmost) PMTBDD for P1+P2+P3.

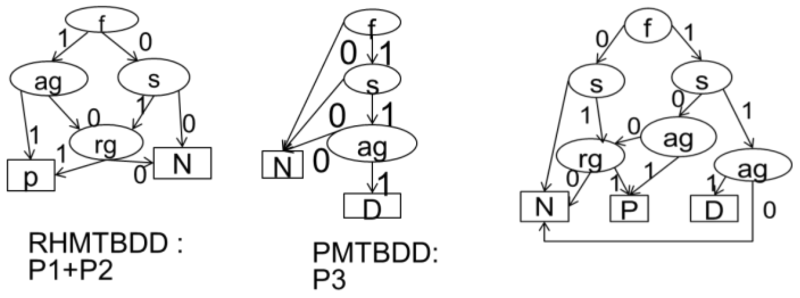


Figure 20: Representing Complex Security Policy.

In general, our future work is to determine whether RHMTBDD can be combined with any arbitrary PMTBDD and come up with a procedure to create positive and negative test cases from the combined PMTBDD.

5.2.2 Integration with JET

The test cases generated from our framework can be given to tool JET [26], [27] to validate the implementation of the code in the context of checking security violation as well as functional violations based on Java Modeling Language specification. JML has many characteristics and its own syntax [28]. JML can be used as a test oracle for the automated validation/verification of Java source code.

REFERENCES

[1] M. Bishop, Introduction to COMPUTER SECURITY, Addison-Wesley, 2004.

[2] "http://docs.oasis-open.org/xacml/cd-xacml-rbac-profile-01.pdf," [Online].

[3] S. Osborn, R. Sandhu and Q. Munawer, "Configuring role-based access control to enforce mandatory and discretionary access control policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 2, pp. 85-106, 2000.

[4] H. Hu and G.-J. Ahn, "Enabling verification and conformance testing for access control model," in *Proceedings of the 13th ACM symposium on Access control models and technologies. ACM*, 2008.

[5] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh and V. Shmatikov, "The most dangerous code in the world: validating ssl," in *Proceedings of the 2012 ACM conference on Computer and communications security. ACM*, 2012.

[6] H. H. Thompson, "Why security testing is hard," *IEEE Security & Privacy*, vol. 1, no. 4, pp. 83-86, 2003.

[7] H. H. Thompson and J. A. Whittaker, How to break software security, Addison Wesley, 2003.

[8] J. F. Kurose and K. W. Ross, Computer Networking A Top-Down Approach Featuring the Internet, Addison Wesley, 2003.

[9] R. S. Sandhu, E. J. Coyne, H. L. Feinstein and C. E. Youman, "Role-based access control models.," in *IEEE computer* , 1996.

[10] J. Kleinberg and E. Tardos, Algorithm Design, PEARSON Addison Wesley, 2006.

[11] K. Fisler, S. Krishnamurthi, L. A. Meyerovich and M. C. Tschantz, "Verification and Change-Impact Analysis of Access-Control Policies," in *In Proceedings of the 27th international conference on Software engineering*, ACM, 2005.

[12] "http://sunxacml.sourceforge.net/," [Online].

[13] "http://docs.oracle.com/javase/tutorial/jaxp/dom/index.html," [Online].

[14] "http://docs.oracle.com/cd/B28359_01/appdev.111/b28394/adx_j_parser.htm#i1013320," [Online].

[15] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker and M. Siegle, "On the use of MTBDDs for performability analysis and verification of stochastic systems," *The Journal of Logic and Algebraic Programming* , vol. 56, no. 1, pp. 23-67, 2003.

[16] E. Clarke, M. Fujita and X. Zhao, "Applications of multi terminal binary decision diagrams (No. CMU-CS-95-160)," Carnegie-Mellon Univ. Pittsburgh PA Dept Of Computer Science., 1995.

[17] M. Fujita, P. C. McGeer and J.-Y. Yang, "Multi-Terminal Binary Decision Diagrams:An Efficient Data Structure for Matrix Representation," *Formal methods in system design* , vol. 10, no. 2-3, pp. 149-169, 1997.

[18] "http://junit.org/," [Online].

[19] "http://www.tutorialspoint.com/junit/," [Online].

[20] "http://docs.oasis-open.org/xacml/3.0/xacml-3.0-rbac-v1-spec-cd-03-en.html," [Online].

[21] "http://msdn.microsoft.com/en-us/library/hh694602.aspx," [Online].

[22] G.-J. Ahn and R. Sandhu, "Role-based authorization constraints specification," *ACM Transactions on Information and System Security (TISSEC),*, vol. 3, no. 4, pp. 207-226, 2000.

[23] H. Hu and G.-J. Ahn, "Constructing Authorization Systems Using Assurance management Framework," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions*, vol. 40, no. 4, pp. 396-405, 2010.

[24] H. Hu, G.-J. Ahn and K. Kulkarni, "Discovery and Resolution of Anomalies in Web Access Control Policies," *IEEE transactions on dependable and secure computing*, vol. 10, no. 6, 2013.

[25] R. Sandhu, F. David and R. Kuhn, "The NIST model for role-based access control: towards a unified standard," in *ACM workshop on Role-based access control*, 2000.

[26] Y. Cheon, "Automated Random Testing to Detect Specification-Code Inconsistencies," in *in Proceedings of the 2007 International Conference on Software Engineering Theory and Practice*, 2007.

[27] C. E. Rubio-Medrano, G.-J. Ahn and K. Sohr, "Verifying Access Control Properties with Design by Contract: Framework and Lessons Learned," in *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference. IEEE Computer Society*, 2013.

[28] L. Burdy, D. R. Cok, Y. Cheon, M. D. Ernst, J. R. Kiniry and G. T. Leavens, "An overview of JML tools and applications," *International journal on software tools for technology transfer,* vol. 7, no. 3, pp. 212-232, 2005.