CERIAS Tech Report 2005-62

MODEL-BASED TESTING OF ACCESS CONTROL SYSTEMS THAT EMPLOY RBAC POLICIES

by Ammar Masood, Rafae Bhatti, Arif Gahfoor, Aditya P. Mathur

This Tech Report was produced jointly with SERC

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Model-based Testing of Access Control Systems that Employ RBAC Policies

Ammar Masood[1], Rafae Bhatti[1], Arif Ghafoor[1] and Aditya Mathur[2]
[1]School of Electrical & Computer Engineering, Purdue University, West Lafayette, IN
[2]Department of Computer Science, Purdue University, West Lafayette, IN
ammar@purdue.edu,bhattir@purdue.edu,ghafoor@ecn.purdue.edu,apm@cs.purdue.edu

Abstract

Access control is the key security service used for information and system security. The access control mechanisms can be used to enforce various security policies, but the desired access control objectives can only be achieved if the underlying software implementation is correct. It therefore becomes essential to not only verify that the implementation conforms to the given policy but also to confirm the absence of any violations in it. We propose a model-based strategy for testing implementations of access control systems that employ the RBAC policy specification. Our approach is based on the construction of a structural and behavioral model of the corresponding RBAC specification. The model is then used to generate static and dynamic test suites for the corresponding implementation. The code coverage and mutation score were used as metrics to determine the efficacy of the proposed approach in a case study. The results of the case study show that the tests generated using the proposed approach not only provide good control flow coverage of the implementation but are also effective in detecting faults induced via mutation operators.

1 INTRODUCTION

Access control is used widely to restrict access to information. It is the key security service providing the foundation for information and system security. Effective use of access control protects the system from unauthorized users [Sandhu94]. The security policy enforced by access control mechanisms could be specified through different representations, such as role based access control (RBAC) [Ferra01], discretionary access control (DAC), and mandatory access control (MAC) [Sandhu94]. Regardless of the specification mechanism used, the desired access control objectives of a system can only be achieved if the corresponding policy specifications are correctly enforced by the underlying software implementation. Hidden functionality, coding errors, wrong configuration, etc. can seriously jeopardize the effectiveness of the corresponding access control mechanism [Thom03]. It therefore becomes essential to assure that the implementation realizes the specified policy correctly and accurately. Such assurance requires the verification of an implementation's conformance to the given

policy, known as security functional testing [Chan04], and also the confirmation of the absence of any violations, known as security vulnerability testing [Du00].

We propose a model-based strategy for testing implementations of access control systems that employ the RBAC policy specification. The model-based testing approach has been adopted as it can be applied to systems developed in house, and to Commercial Off The Shelf (COTS) [Jilani98] products. Through a case study, we show how a finite state model of the access policy specifications can be used to automatically derive tests using a well known algorithm [Chow78]. Tests so derived attained a high degree of control flow coverage of the implementation and, with the aid of program mutation, were found effective in the detection of faults.

Adopting model-based testing allows one to work with software specifications; the source code is not required to perform the test. Using specifications for test generation has several other advantages as compared to code/structure based test generation. As a specification provides the test oracle, a strong relationship exists between the specification and tests that facilitate the location and existence of errors [Stock96]. The test generation process also helps in finding the inconsistencies and ambiguities in the specifications [Offutt03]. Another significant advantage of model-based testing is that tests can be designed earlier in the life cycle of product development allowing for efficient resource allocation.

An implementation is often tested for the presence of security vulnerabilities by execution against a suite of test cases that represent known exploits [Thom03]. This methodology is restricted by the assumption that only known exploits will be used against the product. In contrast, security functional testing is able to verify the conformance to specifications though unable to determine any unspecified or undesired behavior [Dima99]. We therefore argue that a model-based security testing strategy, in which a formal model is used to generate tests, would be complementary to a strategy based only on known exploits. As shown in this work, model-based testing provides a systematic procedure for generating test cases for both security-functional and security-vulnerability testing. The models can be constructed either from the underlying structure of the implementation or from the corresponding access control policy specification; the testing technique can be considered, respectively, as white or black box testing [Beiz90], [Beiz95], [Spence94], [Demillo87].

Contributions

Our contributions are summarized below.

- A method for the construction of structural and behavioral models of RBAC policy specification.

- Automated generation of a test suite from the structural and behavioral models for security testing of the implementation.
- The adequacy and effectiveness assessment of proposed model-based security testing approach through a case study.

The remainder of this paper is organized as follows: Section 2 provides brief introduction to model-based testing techniques; in particular finite state model based testing techniques. The RBAC policy specification is formally defined in Section 3. Construction of structural and behavioral models corresponding to a given RBAC policy specification is discussed in Section 4. Details of using these models to generate a test suite for security testing are also presented in Section 4. In Section 5 we describe a case study using the proposed model-based security testing method to test a prototype RBAC system. We use control flow coverage and program mutation scores to assess the adequacy and fault-detection effectiveness of tests generated automatically. The corresponding results are also presented in Section 5. Section 6 summarizes the results of the case study. The related literature in model-based testing and security testing is discussed in Section 7. Our conclusions and suggested avenues for further research appear in Section 8.

2 MODEL-BASED TESTING USING FINITE STATE MODELS

One approach for applying black box testing uses a formal model of the implementation, usually derived from its requirements, to determine test sequences. The model captures the expected behavior of the implementation. This explicit modeling of the expected behavior makes it simpler and cheaper to perform testing [Beiz95]. A common type of modeling is to represent the product under test by means of finite-state machines (FSM) or state-transition diagrams (or state graphs) [Berto04]. State graphs are a useful way to think about software behavior and testing [Beiz95].

For software designs modeled by a finite state machine (FSM), the W-method is a classic and effective test generation approach [Chow78]. The W-method works by initially generating a test tree from an FSM model and then concatenating the test sequences, generated from the test tree, with the determined state characterization set (W). The resultant test cases generated are able to uniquely identify each leaf node in the test tree. This strategy for test data selection was proved to be both *valid* and *reliable* [Chow78, Good75]. The automata theoretic approach used in the W-method has been used as a basis for other software testing strategies where the software design is modeled by FSMs. The *unique input output* (UIO) [Sabn88] and the *partial W* (W-p) [Fuji91] methods are examples of

methods that evolved out of the W-method. Automatic generation of test cases that satisfy different coverage criteria such as transition coverage and state coverage for *state-based specifications,* has been studied by Offutt et.al. [Offutt03]. The *state-based specifications* imply that the software functional requirements are expressed in terms of states and transitions.

The FSM model of a software design can be viewed as a directed graph with vertices representing the program state and arcs indicating the input/stimuli that change the program state. Each test case consists of a sequence of inputs which when applied to the implementation under test would result in state changes and an expected behavior. The state changes are monitored for verifying the adherence of implementation to its design. The FSM model representing a program can be very huge as the number of states in the FSM grows exponentially [Fried02]. This phenomenon is traditionally referred to as state explosion. The number of states increases as the model attempts to capture more software execution details. State explosion would also result into test cases explosion. One technique to cope with the state explosion problem utilizes a projected state machine model, where a projected state represents a class of states under some equivalence relation [Fried02]. A reduction technique for handling state explosion problem by utilizing structural symmetry information in the system description has been presented in [Ip96]. The test case explosion problem has also been handled by a combinatorial approach in which the generated tests ensure coverage of n-way combinations of the test parameters [Cohen96, Dalal98].

Our focus in this paper is the security testing of access control systems using RBAC policy. In the next section, we provide the details of an RBAC policy specification.

3 RBAC POLICY SPECIFICATION

In RBAC, the access control policy is specified by mapping permissions to roles to which users are assigned. The permissions map the possible authorizations of a role in terms of specific operations that a user activating that role can perform on the corresponding system resource. A user assigned to a role cannot invoke the permissions of that role until the time that user actually activates that role. *Separation of duty* (SoD) is a well-known authorization constraint used in commercial application environments [Ahn2000]. A SoD constraint is intended to prevent a user from acquiring membership of two constrained roles. An RBAC specification provides the rules for user-role assignment (activation) SoD constraints, role hierarchy semantics and static/dynamic user (role) cardinality constraints. A formal definition of RBAC policy specification follows.

Definition 1 ($RBAC_p$): An RBAC policy ($RBAC_p$) is a structure $RBAC_p$={R, U, P, ≤, Status, Permitted, I} such that,

- R={r_1, $r_{2,\ldots\ldots}$ r_n} is a set of n roles where each r_i={Id, c_s, c_d} $1 \leq i \leq n$ is a tuple such that

  (a) Id is the unique identification of r_i

  (b) $c_s \in Z^+$ is the static cardinality i.e. the maximum number of users that can be assigned to this role

  (c) $c_d \in Z^+$ is the dynamic cardinality i.e. the maximum number of users that can activate this role

- U={u_1, $u_{2,\ldots\ldots}u_m$} is the set of m users where u_i={Id,c_s,c_d} $1 \leq i \leq m$ is a tuple such that

  (a) Id is the unique identification of u_i

  (b) $c_s \in Z^+$ is the static cardinality i.e. the maximum number of roles to which this user can be assigned

  (c) $c_d \in Z^+$ is the dynamic cardinality i.e. the maximum number of roles which this user can activate

- P={p_1, $p_{2,\ldots\ldots}p_q$} is a set of q permissions where p_i={Id,obj,op} $1 \leq i \leq q$ is a tuple such that

  (a) Id is the unique identification of p_i

  (b) obj ∈ SystemResources[1] is the system resource/object on which the specified operation op ∈ SystemOperations can be carried out by a user activating such role to which p_i is assigned

- $\leq \subseteq (R \times R)$ where $\leq$={$\leq_A$,$\leq_I$} provides partial ordering on the set of roles such that

  (a) $r_i \leq_A r_j$ means that r_j is senior to r_i as per activation hierarchy (A-hierarchy) semantics [Sandhu98]; a user assigned to r_j can also be able to activate r_i without being actually assigned to it

  (b) $r_i \leq_I r_j$ means that r_j is senior to r_i as per inheritance hierarchy (I-hierarchy) semantics [Sandhu98]; a permission assigned to r_i will also be accessible by r_j without being actually assigned to it

- Status=$UR_{assign} \cup UR_{active} \cup PR_{assign}$ is a set of status predicates partitioned as follows:

  (a) UR_{assign} : U x R → [0|1] where a 1(0) indicates that the given user is assigned (not assigned) to the given role

[1] SystemResources and SystemOperations are system specific sets that represent, respectively, all system resources and all allowed system operations.

(b) UR_{active} : U x R $\rightarrow$ [0|1] where a 1(0) indicates that the given user has activated (not activated) the given role

(c) PR_{assign}: P x R $\rightarrow$ [0|1] where 1 (0) indicates that the given permission is assigned (not assigned) to the given role

- Permitted=$UR_{canAssign} \cup UR_{canActivate} \cup PR_{canAssign}$ is a set of allowable predicates partitioned as follows:

(a) $UR_{canAssign}$: $D_1 \subseteq$ U x R $\rightarrow$ [0|1] where the value of 1 (0) indicates that the given user can be assigned (not assigned) to the given role

(b) $UR_{canActivate}$: $D_2 \subseteq$ U x R $\rightarrow$ [0|1] where a 1(0) indicates that the given user can activate (not activate) the given role

(c) $PR_{canAssign}$: $D_3 \subseteq$ P x R $\rightarrow$ [0|1] where a 1(0) indicates that the given permission can be assigned (not assigned) to the given role

- I={$Assign_{ur}$, $DeAssign_{ur}$, $Activate_{ur}$, $DeActivate_{ur}$, $Assign_{pr}$, $DeAssign_{pr}$} is the set of input requests allowed under the policy such that

(a) $Assign_{ur}(u \in U, r \in R)$ is the input request to assign u to r

(b) $DeAssign_{ur}(u \in U, r \in R)$ is the input request to remove assignment of u to r

(c) $Activate_{ur}(u \in U, r \in R)$ is the input request to allow u to activate r

(d) $DeActivate_{ur}(u \in U, r \in R)$ is the input request to allow u to Deactivate r

(e) $Assign_{pr}(p \in P, r \in R)$ is the input request to assign p to r

(f) $DeAssign_{pr}(p \in P, r \in R)$ is the input request to remove assignment of p to r

As already mentioned, RBAC implements access control decisions by mapping users to roles; permissions are assigned to roles. The access control decisions are guided by formally specified rules. The set of Status and Permitted predicates provided by $RBAC_p$ are used to define the rules that constrain the possible assignments and activations within the given RBAC policy. These rules are provided by the rule set ($\mathfrak{R}$) defined below.

Definition 2 ($\mathfrak{R}$): The rule set $\mathfrak{R}$={$\Upsilon_{urAssignCard}$, $\Upsilon_{urActivationCard}$, Υ_{urSSoD}, Υ_{urDSoD}, Υ_{urHier}, Υ_{prHier}, Υ_1, Υ_2, Υ_3} is the set of system rules that controls the access control decisions in a given $RBAC_p$={R, U, P, $\leq$, Status, Permitted, I}. The rules are given in Table 1.

Table 1: Rules in the rule set $\Re$

Rule	Explanation
$\Upsilon_{urAssignCard}(u\in U, r\in R)=1$ iff $UR_{canAssign}(u,r)=1 \wedge UR_{assign}(u,r)=0 \wedge \sum_{R} UR_{assign}(u,r_i) < c_{s\|u}$ [2] $\wedge \sum_{U} UR_{assign}(u_i,r) < c_{s\|r}$	$\Upsilon_{urAssignCard}$ can only be 1 if the static cardinality constraints corresponding to the given user u and role r are not violated by the assignment of u to r.
$\Upsilon_{urActivationCard}(u\in U, r\in R)=1$ iff $[UR_{canActivate}(u,r)=1 \vee \Upsilon_{urHier}(u, r)=1] \wedge UR_{active}(u,r)=0 \wedge \sum_{R} UR_{active}(u,r_i) < c_{d\|u} \wedge \sum_{U} UR_{assign}(u_i,r) < c_{d\|r}$	$\Upsilon_{urActivationCard}$ can only be 1 if the dynamic cardinality constraints corresponding to the given user u and role r are not violated by the activation of r by u.
$\Upsilon_{urSSoD}(u\in U, r\in R)=1$ given $c_{ssod}\leq\|R'\cup r\|$ iff $\sum_{R'} UR_{assign}(u,r_i) < c_{ssod}$	where R′ is the static SoD (SSoD) set corresponding to r and c_{ssod} is the cardinality of SSoD set i.e. the maximum number of roles to which u can be simultaneously assigned in the set $R'\cup r$. Υ_{urSSoD} can only be 1 if user u can be assigned to r such that the total number of user-role assignments corresponding to u in the set R′ are less than c_{ssod}.
$\Upsilon_{urDSoD}(u\in U, r\in R)=1$ given $c_{dsod}\leq\|R'\cup r\|$ iff $\sum_{R'} UR_{active}(u,r_i) < c_{dsod}$	where R′ is the dynamic SoD (DSoD) set corresponding to r and c_{dsod} is the cardinality of DSoD set i.e. the maximum number of roles in the set $R'\cup r$ which can be concurrently activated by u. Υ_{urDSoD} can only be 1 if user u can activate r such that the total number of user-role activations corresponding to u in the set R′ are less than c_{dsod}.
$\Upsilon_{urHier}(u\in U, r\in R)=1$ iff $UR_{active}(u,r)=0 \wedge \exists r'\in R'\| R'\subseteq R \wedge UR_{assign}(u, r')=1$	where $R':\{r'\| r\leq_A r'\}$. R′ is the set of all roles senior to r as per A-hierarchy semantics (r is also member of this set). $\Upsilon_{urHier}(u,r)=1$ implies that there is at least one such role r′(could be r), senior to r, to which u is currently assigned. A-hierarchy semantics thus permit activation of a junior role by the user provided that the user is assigned to at least one role senior to former.
$\Upsilon_{prHier}(p\in P, r\in R)=1$ iff $\exists r'\in R'\| R'\subseteq R \wedge PR_{assign}(p, r')=1$	where $R':\{r'\| r' \leq_I r\}$. R′ is the set of all roles junior to r as per I-hierarchy semantics (r is also member of this set). $\Upsilon_{prHier}(p,r)=1$ implies that there is at least one such role r′(could be r), junior to r, to which p is currently assigned. I-hierarchy semantics thus permit assignment of permissions to a senior role on the basis of there being assigned to a junior role.
Υ_1: $Assign_{ur}(u\in U, r\in R)\Rightarrow Update_{status}$ [3] $[UR_{assign}(u,r)=1] \wedge Update_{permitted} [UR_{canActivate}(u,r)=1]$ iff $\Upsilon_{urAssignCard}(u, r)=1 \wedge \Upsilon_{urSSoD}(u, r)=1$ and $DeAssign_{ur}(u\in U, r\in R)\Rightarrow Update_{status} [UR_{assign}(u,r)=0] \wedge Update_{status} [UR_{active}(u,r)=0]$	This rules ensures that the user-role assignment corresponding to the input $Assign_{ur}(u,r)$ is allowed only if the user/role static cardinality constraints and role SSoD constraints are not violated by such assignment[4].
Υ_2: $Activate_{ur}(u\in U, r\in R)\Rightarrow Update_{status} [UR_{active}(u,r)=1]$ iff $\Upsilon_{urActivationCard}(u, r)=1 \wedge \Upsilon_{urDSoD}(u, r)=1$ and $DeActivate_{ur}(u\in U, r\in R)\Rightarrow Update_{status} [UR_{active}(u,r)=0]$	This rule ensures that the user-role activation corresponding to the input $Activate_{ur}(u,r)$ is allowed only if the user/role dynamic cardinality constraints, role DSoD/A-hierarchy constraints are not violated by such activation.
Υ_3: $Assign_{pr}(p\in P, r\in R)\Rightarrow Update_{status} [PR_{assign}(p,r)=1]$ iff $PR_{canAssign}(p,r)=1 \vee \Upsilon_{prHier}(p, r)=1$and $DeAssign_{pr}(p\in P, r\in R)\Rightarrow Update_{status} [PR_{assign}(p,r)=0]$	This rules ensures that the permission-role assignment corresponding to the input $Assign_{pr}(p,r)$ is done only either if such assignment is permitted directly in the policy or is allowed by virtue of I-hierarchy semantics.

[2] $c_{s|u}$ ($c_{s|r}$) indicates static cardinality c_s corresponding to u (r). The dynamic cardinality is also referenced similarly.

[3] $Update_{status}[x\in Status=0/1]$ implies that assignments/activations in the current RBAC state are updated so that the current value of the corresponding predicate becomes 0/1, $Update_{permitted}$ denotes update of permitted predicate to the new value

[4] The effect of complementary inputs e.g. $DeAssign_{ur}$ is obvious and is thus not further explained

It is to be noted that in $\Re$, all the access control decisions are actually enforced by the three final rules Υ_1, Υ_2 and Υ_3 while the remaining rules serve to support the final rules. The following example elucidates security testing and illustrates a few of the rules mentioned above.

Example 1: Consider an RBAC policy specification involving users u_1, u_2 and roles r_1, r_2, r_3. Suppose that u_1 is assigned to r_1 and r_2, and u_2 is assigned to r_2 and r_3. Further, there is a SoD constraint on the activation of r_1 and r_2. Subject policy is required to be enforced by an implementation. In this case, security functional testing would be carried out to ensure the conformity of the implementation to its specifications e.g. user u_2 is always able to activate r_2, user u_1 cannot violate the SoD constraint and be able to activate r_1 and r_2 simultaneously, etc. However, security vulnerability testing would test the implementation for the presence of any undesired behavior, which in this case could be to test whether or not it is possible for user u_1 to become assigned to role r_3 without any provision for such assignment in the policy. The different components of the corresponding $RBAC_p$ are:

$U= \{u_1, u_2\}$ where $u_1= \{u_1, 2, 1\}$ and $u_2= \{u_2, 2, 2\}$

$R=\{r_1, r_2, r_3\}$ where $r_1=\{r_1, 2, 1\}$, $r_2=\{r_2, 2, 2\}$, $r_3=\{r_3, 1, 1\}$ and the DSoD set R' corresponding to $r_2(r_1)=\{ r_1\}(\{r_2\})$ with $c_R=1$

Consider $P= \{p_1, p_2\}$, where $p_1= \{p_1, file, read\}$ and $p_2= \{p_2, file, write\}$

The initial value of Permitted and Status predicates is shown in Table 2. We assume that the set of inputs as given in Table 2 is already applied to the system when a decision is to be made regarding the input $Activate_{ur}(u_1, r_2)$. At that time the values of Permitted and Status predicates would have changed from their initial value and are given by the "Final Value" in the table.

Table 2: Example 2

Initial Value			Final Value	
Permitted predicates	Status predicates	Inputs	Permitted predicates	Status predicates
$UR_{canAssign}(u_1, r_1)=1$	$UR_{assign}(u_1, r_1)=0$	$Assign_{ur}(u_1, r_1)$	$UR_{canAssign}(u_1, r_1)=1$	$UR_{assign}(u_1, r_1)=1$
$UR_{canAssign}(u_1, r_2)=1$	$UR_{assign}(u_1, r_2)=0$	$Assign_{ur}(u_1, r_2)$	$UR_{canAssign}(u_1, r_2)=1$	$UR_{assign}(u_1, r_2)=1$
$UR_{canAssign}(u_2, r_2)=1$	$UR_{assign}(u_2, r_2)=0$	$Assign_{ur}(u_2, r_2)$	$UR_{canAssign}(u_2, r_2)=1$	$UR_{assign}(u_2, r_2)=1$
$UR_{canAssign}(u_2, r_3)=1$	$UR_{assign}(u_2, r_3)=0$	$Assign_{ur}(u_2, r_3)$	$UR_{canAssign}(u_2, r_3)=1$	$UR_{assign}(u_2, r_3)=1$
$UR_{canActivate}(u_1, r_1)=0$	$UR_{active}(u_1, r_1)=0$	$Activate_{ur}(u_1, r_1)$	$UR_{canActivate}(u_1, r_1)=1$	$UR_{active}(u_1, r_1)=1$
$UR_{canActivate}(u_1, r_2)=0$	$UR_{active}(u_1, r_2)=0$		$UR_{canActivate}(u_1, r_2)=1$	$UR_{active}(u_1, r_2)=0$
$UR_{canActivate}(u_2, r_2)=0$	$UR_{active}(u_2, r_2)=0$		$UR_{canActivate}(u_2, r_2)=1$	$UR_{active}(u_2, r_2)=0$
$UR_{canActivate}(u_2, r_3)=0$	$UR_{active}(u_2, r_3)=0$		$UR_{canActivate}(u_2, r_3)=1$	$UR_{active}(u_2, r_3)=0$
$PR_{canAssign}(p_1,r_1)=1$	$PR_{assign}((p_1,r_1)=0$		$PR_{canAssign}(p_1,r_1)=1$	$PR_{assign}((p_1,r_1)=0$

By rule Υ_2, which decides the outcome of $Activate_{ur}(u_1, r_2)$, u_1 will be able to activate r_2 only if $\Upsilon_{urActivationCard}(u_1, r_2)=1 \wedge \Upsilon_{urDSoD}(u_1, r_2)=1$. However, in this case, as $\Upsilon_{urDSoD}(u_1, r_2)=0$, the requested activation will not be allowed.

As mentioned in Section 1, a software implementation is expected to fully enforce the access control policy specified by $RBAC_p$. However, due to implementation errors, the policy may not be correctly enforced. The purpose of security testing is to check the behavior exhibited by an implementation for conformance to the expected behavior specified by $RBAC_p$. We envisage the following sequential steps, shown graphically in Figure 1, that lead to a fully tested implementation.

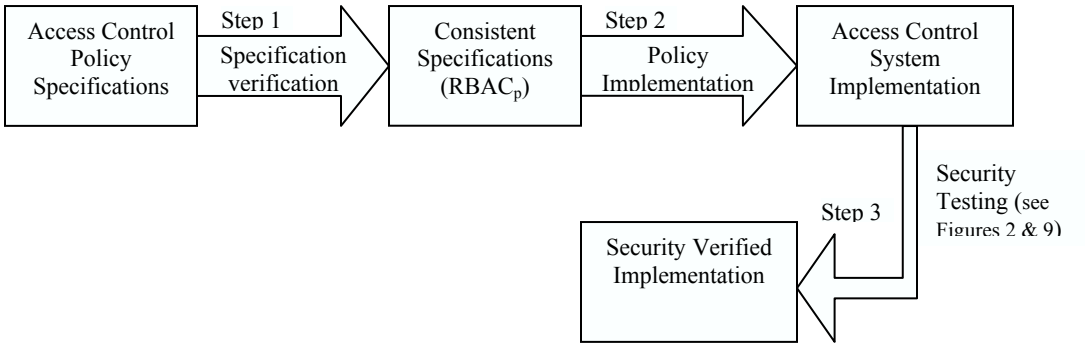

Figure 1: Sequential steps leading to a verified implementation

The first step in obtaining a security verified implementation is to verify the access control system specifications for consistency. The result of specification verification is a consistent specification ($RBAC_p$[5]) shown to be free of any conflicts. In step 2, $RBAC_p$ is realized by the underlying software implementation. Finally in step 3, the security testing is carried out, which is also the scope of this paper, to validate the implementation with respect to $RBAC_p$.

4 MODEL-BASED SECURITY TESTING

We now propose a model-based strategy for security testing where $RBAC_p$ is used directly to construct the expected structural and behavioral models of the implementation. These models serve as inputs to the test generation algorithm to generate the static and dynamic test suites (Figure 2). The test suites are executed against the implementation and the results are correlated with the models. The advantages of using models, directly based on $RBAC_p$, are two-pronged. First it offers a systematic procedure to generate test suites directly from specifications and secondly, as already discussed in Section 2, proven techniques exist for generating tests from suitable models, which can also be leveraged for our current application.

[5] We assume that $RBAC_p$ is free of all inconsistencies and conflicts and is thus considered a verified policy. We thus do not consider the problem of policy verification which has been addressed previously [Ahmed03, Lupu99]

In carrying out security testing of the implementation, the goal would ideally be to exhaustively perform both the security functional and security vulnerability testing. However, as already mentioned, vulnerability testing is limited by the extent of knowledge about the known exploits against the system. While the proposed model-based approach for vulnerability testing remains restricted by the extent of the implementation details captured by the model, it provides a systematic way to generate test cases that directly correlate with the specifications.

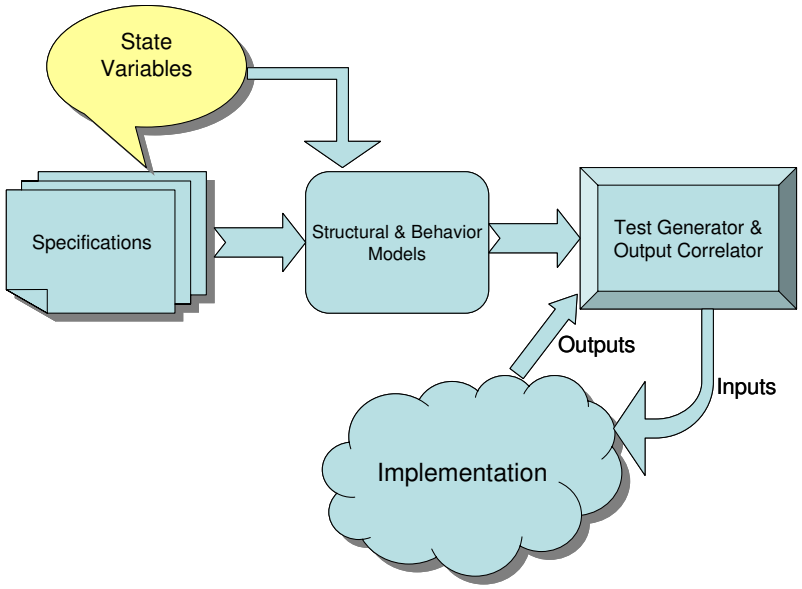

Figure 2: Model based Security Testing

We believe that security testing of access control systems can be performed under two different paradigms: developer-oriented and user-oriented. In developer-oriented paradigm, the focus is on the verification of the implementation with respect to the possible constraints/rules in the policy specification. In the user-oriented paradigm, the focus is directed towards the verification of the implementation within the constraints of the user operational domain. In the latter case, the end user performs security testing to assure the effectiveness of the given access control system with respect to the access control policies within the user's expected operational environment. As compared to user-oriented testing where $RBAC_p$ would be readily available as a domain specification, in developer-oriented testing, an additional requirement would be to generate suitable $RBAC_p$ models to ensure complete testing of the system. Our model-based security testing approach is equally applicable to both the paradigms.

4.1 Structural Model

The structural model of the implementation includes all possible user-role and permission-role pairs in $RBAC_p$. Hereafter, user-role and permission-role are abbreviated as UR and PR, respectively. It is important to capture this information as $RBAC_p$ might not convey information about all the possible UR and PR assignments (i.e. $D_1 \subseteq UxR$ and $D_3 \subseteq PxR$, as per Definition 1) whereas all ambiguities must be resolved for verification of the implementation. The structural model is defined below.

Definition 3 (M_{struct}): The structural model (M_{struct}) corresponding to $RBAC_p$ is a tuple $M_{struct}=\{UR_{struct}, PR_{struct}\}$ such that,

- $UR_{struct}=\{ur_{struct} | ur_{struct} \in U \text{ x } R\}$ is the set of ur_{struct} elements where $ur_{struct}=\{u_i, r_j\}$ $1 \leq i \leq m$, $1 \leq j \leq n$. Hence $|UR_{struct}|=m \text{ x } n$
- $PR_{struct}=\{pr_{struct} | pr_{struct} \in P \text{ x } R\}$ is the set of pr_{struct} elements where $pr_{struct}=\{p_i, r_j\}$ $1 \leq i \leq q$, $1 \leq j \leq n$. Hence $|PR_{struct}|=q \text{ x } n$

The static test suite, generated from the structural model, considers all ambiguous assignments not to be allowed, and thus is also able to support vulnerability testing by looking for possible exploits in the UR and PR assignments. One such exploit of u_1 being able to get assigned to r_3 was discussed in Example 1. The static test suite is generated, and the implementation is executed against it, using algorithm $Run_{structTest}$ below. In $Run_{structTest}$, the actual output/response of the implementation for all inputs is referenced by attaching the prefix "Out" to the corresponding input e.g $OutAssign_{ur}(ur_{struct})$ denotes the output corresponding to the input $Assign_{ur}(ur_{struct})$.

The algorithm $Run_{structTest}$, takes the structural model M_{struct} and $RBAC_p$ as inputs and applies the UR (PR) assignment inputs, corresponding to all the UR_{struct} (PR_{struct}) elements of M_{struct}, to the implementation. The results of the assignment operations are compared with the desired values determined by the application of rule Υ_1 or Υ_3 on the corresponding input. If the former agrees with the latter then the test passes, else it fails. The PR assignment tests are also able to verify the correctness of the implementation of the I-hierarchy. It is to be noted that the de-assignment inputs are also applied to the implementation to ensure the conformance of its structural model to $RBAC_p$.

Example 2: For the $RBAC_p$ in Example 1, the corresponding M_{struct} is:

$UR_{struct}=\{u_1r_1, u_1r_2, u_1r_3, u_2r_1, u_2r_2, u_2r_3\}$, $PR_{struct}=\{p_1r_1, p_1r_2, p_1r_3, p_2r_1, p_2r_2, p_2r_3\}$

In the algorithm $Run_{structTest}$, the *for* loop at lines 2-8 tests the implementation against all the members of UR_{struct}. As per $RBAC_p$, all these assignments should be valid except for u_1r_3 and u_2r_1. Any

mismatch between the implementation and $RBAC_p$ would lead to a failed test. Similarly, the *for* loop at lines 9-15 tests the implementation for all the de-assignment operations corresponding to M_{struct}.

The *for* loop at lines 17-23 (and at lines 24-30) tests the implementation for conformance to $RBAC_p$ relative to the expected and desired outcome of the operations on members of PR_{struct}.

```
Algorithm Run_structTest
Input: M_struct, RBAC_p
Output: URresult={urres_1, urres_2, ……urres_m x n},
PRresult={prres_1, prres_2, ……prres_q x n}

1   i ←1, j ←1
2   for each ur_struct ∈ UR_struct
3      do apply input Assign_ur(ur_struct)
4      apply rule ϒ1 on Assign_ur(ur_struct) to determine
       UR_assign(ur_struct)
5      do if OutAssign_ur(ur_struct)= UR_assign(ur_struct)
6         then urres_i ← "pass"
7      else urres_i ← "fail"
8      i ← i+1
9   for each ur_struct ∈ UR_struct
10     do apply input DeAssign_ur(ur_struct)
11     apply rule ϒ1 on DeAssign_ur(ur_struct) to determine
       UR_assign(ur_struct)
12     do if OutDeAssign_ur(ur_struct)= UR_assign(ur_struct) ∧
       urres_j = "pass"
13        then urres_j ← "pass"
14     else urres_j ← "fail"
15     j ← j+1
16  i ←1, j ←1
17  for each pr_struct ∈ PR_struct
18     do apply input Assign_pr(pr_struct)
19     apply rule ϒ3 on Assign_pr(pr_struct) to determine
       PR_assign(pr_struct)
20     do if OutAssign_pr(pr_struct)= PR_assign(pr_struct)
21        then prres_i ← "pass"
22     else prres_i ← "fail"
23     i ← i+1
24  for each pr_struct ∈ PR_struct
25     do apply input DeAssign_pr(pr_struct)
26     apply rule ϒ3 on DeAssign_pr(pr_struct) to determine
       PR_assign(pr_struct)
27     do if OutDeAssign_pr(pr_struct)= PR_assign(pr_struct) ∧
       prres_j ← "pass"
28        then prres_j ← "pass"
29     else prres_j ← "fail"
30     j ← j+1
31  return URresult, PRresult
```

Figure 3: Procedure for running the static test suite.

4.2 Behavior Model

As already noted, RBAC is used to manage a user's ability to exercise permissions through the mechanism of UR activations. Access to system resources is allowed only if the user activates a suitable role to which the corresponding permission for desired access is assigned, or authorized, as per the I-hierarchy. Thus the primary source of exploits in the implementation would be the user's ability to activate unauthorized roles. It therefore becomes important to not only completely test the implementation for conformance of UR activations to the given $RBAC_p$, but also to verify that the implementation does not allow any exploits, i.e. allow users to activate unauthorized roles not specifically constrained by the policy.

We therefore construct the behavior model (M_{behav}) that represents an implementation's desired response to all possible sequences of UR activation inputs ($Activate_{ur}$). The behavior model is essentially an FSM representing the expected behavior of the implementation corresponding to the applied inputs. While constructing M_{behav}, it is assumed that M_{struct} has been verified through the

execution of implementation against the static test suite, UR_{struct} is then modified to generate UR_{behav} defined below:

Definition 4 (UR_{behav}): It is obtained by trimming UR_{struct} such that $UR_{behav}=\{ur_{struct} | ur_{struct} \in UR_{struct} \wedge [(urres(ur_{struct})=pass \wedge Assign_{ur}(ur_{struct})=1) \vee \Upsilon_{urHier}(ur_{struct})=1]\}$ where $urres(ur_{struct})$ indicates the value of $urres_i$ corresponding to the specific ur_{struct} element.

The ur_{stuct} would be a member of UR_{behav} either if $urres(ur_{struct})=pass \wedge UR_{assign}(ur_{struct})=1$ which would be true if after performing the test using the static test suite, the corresponding UR assignment is found valid in the system, or $\Upsilon_{urHier}(ur_{struct})=1$. $\Upsilon_{urHier}(ur_{struct})=1$ would be true if corresponding to $ur_{struct}=(u,r)$, there is a $ur'_{struct}=(u,r')$ such that $UR_{assign}(ur'_{struct})=1$, and $r' \in R'$ (as in $\Upsilon_{urHier}(ur_{struct})$ definition). This condition would thus be true if the UR activation corresponding to the given ur_{struct} element is allowed as per A-hierarchy semantics. $UR_{assign}(ur'_{struct})=1$ should now be valid as the static test suite has already been used to verify the implementation.

It is to be noted that $UR_{behave} \subseteq UR_{struct}$, because UR_{behave} includes only such elements of UR_{struct} for which the above definition holds. This is essential to reduce the number of state variables in the model (see Definition 5); however, this would also limit the ability of a dynamic test suite to identify such exploits where a user can activate a role without even being assigned to it or to some senior role in the A-hierarchy. In order to overcome this shortcoming, we suggest the static validation of all $ur_{struct} \notin UR_{behave}$ by verifying that corresponding to the application of input $Activate_{ur}(ur_{struct})$ and rule Υ_2, $OutActivate_{ur}(ur_{struct})=0= UR_{active}(ur_{struct})$, i.e. the specific UR activation is also not permitted in the implementation.

4.2.1 Model Construction

In M_{behav}, a set of state variables characterizes the system state. Each state variable corresponds to a UR pair and represents activation/deactivation of the given role by the corresponding user. Starting from an initial state, the system state would change in response to the application of UR activation/assignment (deactivation/deassignment) inputs. Without any constraints on the allowable set of inputs, the state model would explode. We make the following assumptions to avoid state explosion while constructing M_{behav}.

(a) Users are assigned to the roles initially and no deassignment takes place dynamically.

(b) A role cannot be deactivated after a user has activated a role. This restriction allows for the identification of maximal states in M_{behav}, i.e. the final states in the model. Without this assumption the length of input sequences could be infinite.

(c) Once a particular UR activation has occurred, the corresponding input, which led to that activation, is not available until the time the given activation terminates. This is a valid assumption for access control implementations as in practice a logged user cannot login again in the same system.

Definition 5 (M_{behav}): The behavior model (M_{behav}) of $RBAC_p$ is a tuple $M_{behav}=\{ S, I_b, S', \delta\}$ such that,

- $S=\{s_1, s_2, \ldots..,s_t\}$ is a finite set of t states such that $s_i=\{UR_{active}(ur_{struct})| ur_{struct} \in UR_{behav}\}$ $1 \leq i \leq t$, is a set of status predicates corresponding to all elements of UR_{behav}. Further, $s_1=\{ 0,0,0,\ldots.0\}$ is the initial state i.e. initially there are no UR activations in the implementation.
- $I_b=\{Activate_{ur}(ur_{struct})| ur_{struct} \in UR_{behav}\}$ is the set of activation inputs corresponding to all such ur_{struct} elements which are also member of UR_{behav}.
- $S' \subseteq S \times I_b$, if $S' = S \times I_b$ then M_{behav} is fully specified; however as already discussed in assumption (c), M_{behav} will always be partially specified.
- $\delta: S' \rightarrow S$ is the state transition function

In algorithm $Construct_{behav}$ (Figure 4) used for the construction of M_{behav}, $I_b|s$ represents the set of activation inputs available in state s, which would be constrained by assumption (c) discussed above. After creating the initial state, procedure $generate_{nextState}$ is called recursively until all possible states are visited using depth first traversal. In $generate_{nextState}$, corresponding to each application of input $Activate_{ur}(ur_{struct})$, there could be two possible cases for the next state. First, the next state is the same as the current state, which would be true if the corresponding UR activation cannot be made due to the violation of constraints as identified by Υ_2. Second, the algorithm would recursively traverse a different next state. It is easy to conclude that the algorithm would terminate as there are only a finite number of possible states.

```
Algorithm Construct_behav
Input: UR_behav, RBAC_p
Output: M_behav
1   s_1 ← {0,0,0,….0} where |s_1|=|UR_behav|
2   generate_nextState(s_1, I_b|s_1)

generate_nextState(s, I_b|s)
3   for each Activate_ur(ur_struct)∈ I_b|s
4      do apply rule Υ_2 on Activate_ur(ur_struct)
5    to determine s'←{s | UR_active(ur_struct)}
6      δ(s, Activate_ur(ur_struct)) ← s'
7      do if s' ≠ s
8         then generate_nextState(s', I_b|s')
8 end for
s | URactive(urstruct) indicates new value of s
with updated value of URactive(urstruct)
```

Figure 4: Procedure for constructing the behavior model

Corresponding to Example 2, $UR_{behav}=\{u_1r_1, u_1r_2, u_2r_2, u_2r_3\}$ and the resultant M_{behav} generated using $Construct_{behav}$ is shown in Figure 5.

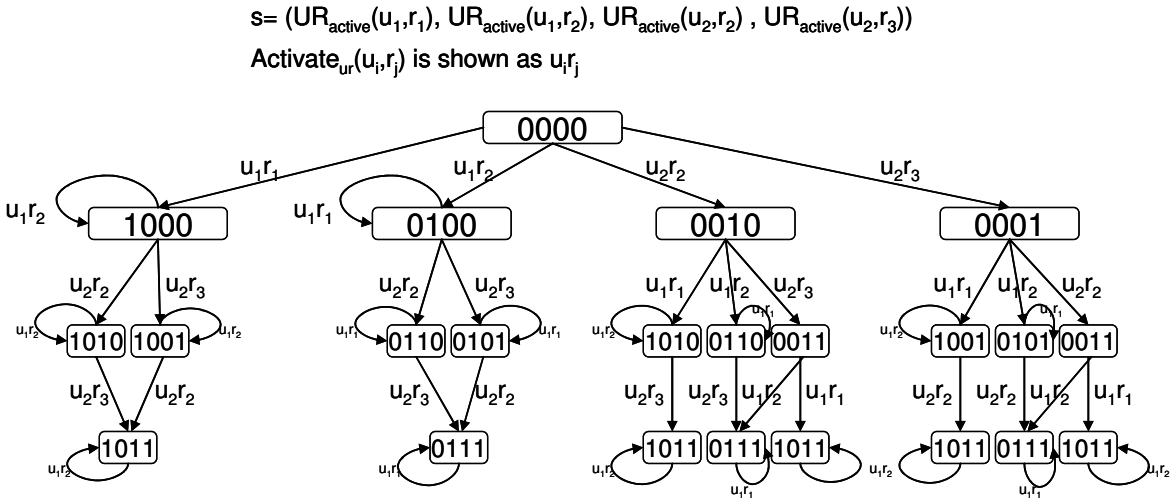

Figure 5: M_{behav} corresponding to Example 2

4.2.2 Dynamic Test Suite

The dynamic test suite is constructed from M_{behav} which is effectively an FSM representation. The W-method is a classic test generation method [Chow78] to generate tests from an FSM specification. Due to the proven fault detection effectiveness of test generated using the W method; we decided to use it for generating the dynamic test suite. As mentioned earlier, the W-method works by first generating a *test tree* from the FSM model and then concatenating the test sequences (generated from the test tree) with the determined state characterization set (W).

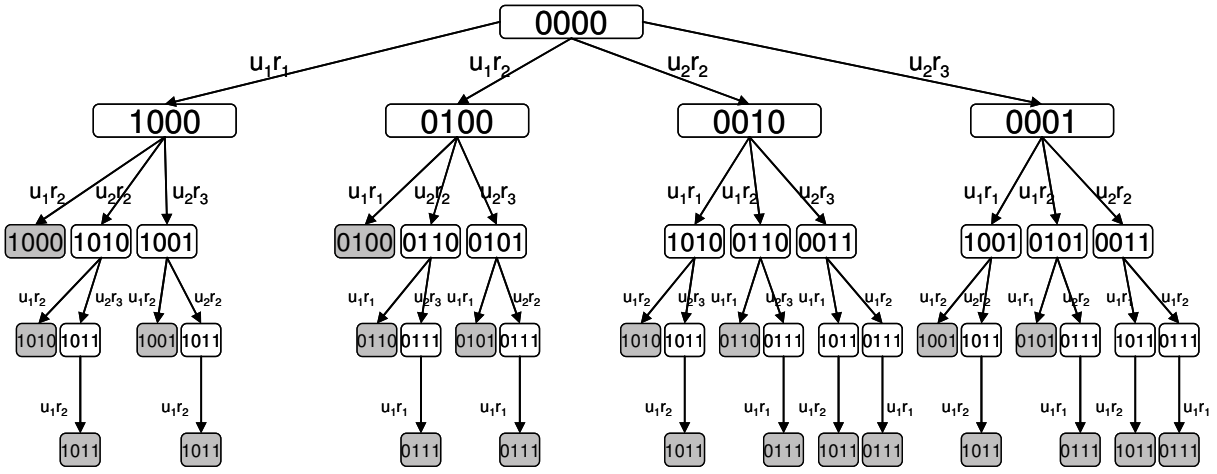

Figure 6: Test Tree corresponding to M_{behav} of Example 2

The W set consists of input sequences that can distinguish between behaviors of every pair of states in the minimal FSM. We assume the existence of such reliable methods in the implementation that can be used to directly check the state. Under this assumption, the states of M_{behav} are readily observable and therefore the W set is not required. Such an assumption has also been used in the Binder round trip method [Binder99] for class testing in object-oriented programs. The dynamic test suite is thus constructed from the *test tree*; the W set is not required. In constructing the *test tree* from the FSM, if the added node at depth k is the same as some other node at depth i, i ≤k, then that node is terminated with no further edge out of it [Chow78]. The *test tree* corresponding to M_{behav} of Figure 5 is shown in Figure 6. The dynamic test suite is constructed from the *test tree* and executed on the implementation via algorithm $Run_{dynamicTest}$ given in Figure 7.

In the algorithm $Run_{dynamicTest}$, n_{node} is the next node in the path ρ and is obtained through the function $node_{next}$ that returns the next node in the path, to which the implementation state is compared after applying the input $Activate_{ur}|$ (ρ,$current_{node}$, n_{node}). The input $Activate_{ur}|$ (ρ,$current_{node}$, n_{node}) denotes the input on the edge between $current_{node}$ and n_{node} in the path ρ in the *test tree*. On reaching the last node of ρ, the $DeActivate_{ur}$ inputs are applied in reverse order to the $Activate_{ur}$ inputs so as to take the implementation back to its initial state i.e. no UR activations exist in the system. Applying $Run_{dynamicTest}$ on the *test tree* of Figure 6, it can be observed that one test sequence would be u_1r_1, u_2r_2, u_1r_2, $u_1r'_2$, $u_2r'_2$, $u_1r'_1$, where $u_ir'_j$ denotes $DeActivate_{ur}(u_i,r_j)$.

```
Algorithm Run_dynamicTest
Input: TestTree
Output: result ∈ {pass, fail}

1   n_0 ← node_root, current_node ← n_0, n_node ← null
2   for each path ρ∈ TestTree
3     do n_node ← node_next
4     do while n_node ≠ null
5          do apply input Activate_ur| (ρ,current_node, n_node)
6          do if current implementation state ≠ n_node
7             then result ← "fail"
8             return result
9          else current_node← n_node
10    end while
11    do apply DeActivate_ur inputs in reverse order
12  to take implementation back to initial state
13   end for
14  result ← "pass"
15  return result
```

Figure 7: Procedure for constructing and executing the dynamic test suite

The total number of test sequences can be reduced by constructing a modified *test tree* where instead of terminating a path at a repeated node, i.e. at a state with a self edge in M_{behav}, the repeated nodes, whenever encountered, are included explicitly in the path. This can be easily observed in the modified *test tree* shown in Figure 8. In our case study (Section 5) we used the original *test tree* for constructing the dynamic test suite.

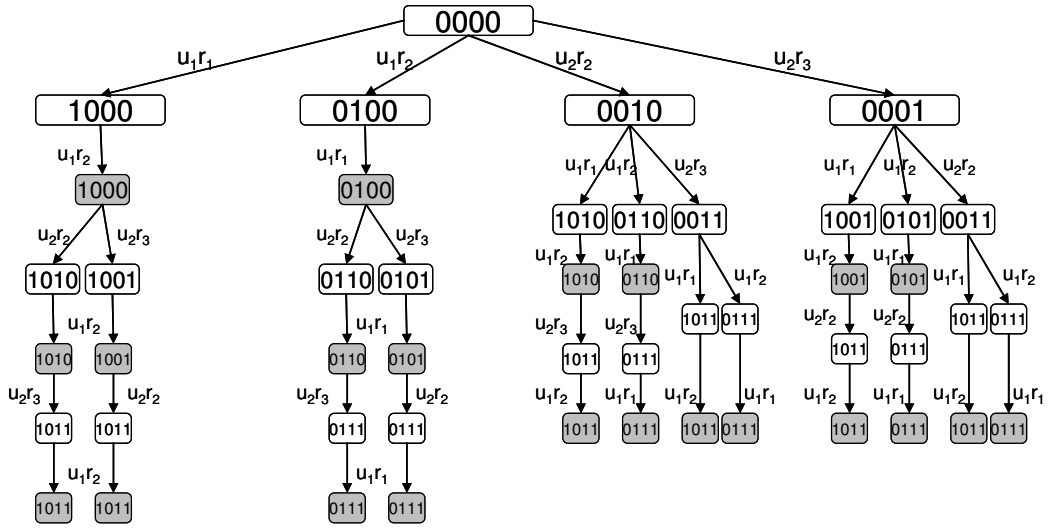

Figure 8: Modified Test Tree corresponding to M_{behav} of Example 2

The complexity of the algorithm $Run_{dynamicTest}$ primarily depends on the complexity of the W-method in construction of the *test tree* which is bounded by product of the number of states (in this case $2^{|UR_{behave}|}$) and the number of input symbols ($|UR_{behave}|$). However, by virtue of our experience with the case study conducted in Section 5, it was observed that for most of the developer-oriented testing, the number of states is not very high except for the tests related to "Role Activation Constraints" (Table 4) for which we reduced the number of test cases by only traversing the *test tree* to depth 1. For a large number of state variables, one solution could be to use combinatorial testing strategies, such as covering arrays, to reduce the number of test cases [Dalal98].

5 CASE STUDY

In this section we present a case study in which the proposed model-based security testing approach was used to verify a variant[6] of the X-GTRBAC prototype system [Bhatti05]. The central idea is that the system uses credentials supplied by users to assign them to roles subject to any assignment constraints. The users can then access resources according to their role memberships subject to any dynamic access constraints. The X-GTRBAC is an XML based system in which $RBAC_p$ is specified using XML policy files. The basic policy files are shown in Table 3 and the system architecture, with added security testing module, is given in Figure 9. Details of the X-GTRBAC system are described elsewhere [Bhatti05].

[6] We considered the variant without temporal constraints

Table 3: X-GTRBAC Policy Files [Bhatti05]

File Name	Purpose
XML User Sheet (XUS)	Declares the users and their authorization credentials
XML Role Sheet (XRS)	Declares the roles, their attributes, role hierarchy, and any separation of duty and temporal constraints associated with roles
XML Permission Sheet (XPS)	Declares the available permissions
XML User-to-Role Assignment Sheet (XURAS)	Defines the rules for assignment of users to roles; these assignments may have associated temporal constraints
XML Permission-to-Role Assignment Sheet (XPRAS)	Defines the rules for assignment of permissions to roles; these assignments may have associated temporal constraints
XML Separation Of Duty Definition Sheet (XSoDDef)	Defines the separation of duty constraints on roles
XML Credential Type Definition Sheet (XCredTypeDef)	Defines the available credential types

5.1 X-GTRBAC System Security Test Process

The testing of X-GTRBAC system is performed through developer-oriented security testing for which suitable $RBAC_p$ models are generated to completely test the system as mentioned in Section 4. This is achieved by generating a suitable set of policy files corresponding to the requirement of each $RBAC_p$ model. As shown in Figure 9, the test policies generated by the security testing module act as input to both the system and the model generator. The model generator creates both the structural and behavior models from the corresponding $RBAC_p$, and also constructs the respective test suites. The test vectors sub-module runs both the static and dynamic suites on the system using the X-GTRBAC system application programmer interface (API). Test results are compared with the expected outputs to verify the system function.

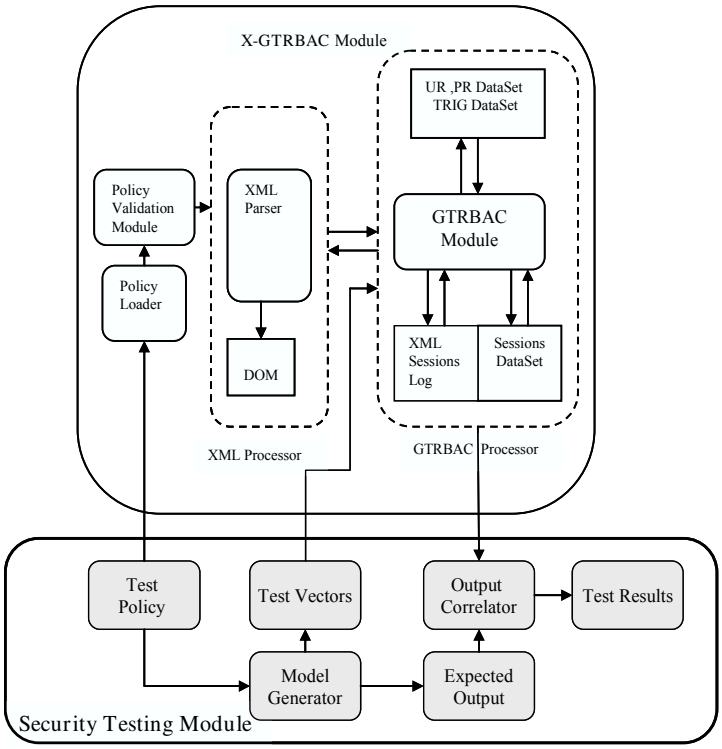

Figure 9: X-GTRBAC System + Security Testing Module

5.2 Constructing Test Policies

By using the test policy files, we are able to verify the ability of the X-GTRBAC system to properly parse the XML files and correspondingly correctly implement the relevant $RBAC_p$. Different policy files are composed depending on the particular aspect of the X-GTRBAC system to be verified.

Table 4: Policy Files Corresponding to Test Objectives

Test objectives	XUS	XRS	XPS	XURAS	XPRAS	XSoDDef
User Cardinality	# user=1 User card=k Add Cred=true	# role=k+1 Role card=1 Hierarchy=false Activ const=false	# perms=p	# user=1 # not Assigned=0 # role=k+1	# perms=p # role=k+1	DSoD=null SSoD=null
Role Cardinality	# user=k+1 User card=1 Add Cred=true	# role=1 Role card=k Hierarchy=false Activ const=false	# perms=p	# user=k+1 # not Assigned=0 # role=1	# perms=p # role=k	DSoD=null SSoD=null
DSoD	# user=1 User card=k+1 Add Cred=true	# role=k+1 Role card=1 Hierarchy=false Activ const=false	# perms=p	# user=1 # not Assigned=0 # role=k+1	# perms=p # role=k+1	DSoD card=k SSoD=null
SSoD	# user=1 User card=k+1 Add Cred=true	# role=k+1 Role card=1 Hierarchy=false Activ const=false	# perms=p	# user=1 # not Assigned=0 # role=k+1	# perms=p # role=k+1	DSoD=null SSoD card=k
Role Hierarchy	# user=1 User card=k Add Cred=true	# role=k Role card=1 Hierarchy=true Activ const=false	# perms=p	# user=1 # not Assigned=0 # role=1	# perms=p # role=1	DSoD=null SSoD=null
User-Role Assignment	# user=k User card=k Add Cred=true	# role=k Role card=k Hierarchy=false Activ const=false	# perms=p	# user=k # not Assigned=k/2 # role=k	# perms=p # role=k	DSoD=null SSoD=null
User-Role Assignment Empty	# user=k User card=k Add Cred=false	# role=k Role card=k Hierarchy=false Activ const=false	# perms=p	No entries made	No entries made	DSoD=null SSoD=null
Role Activation Constraints	# user=1 User card=k Add Cred=true	# role=k Role card=1 Hierarchy=true Activ const=true	# perms=p	# user=1 # not Assigned=0 # role=k	# perms=p # role=k-1	DSoD=null SSoD=null

Table 4 lists the details for each policy file created to verify a particular aspect of the system. XCredTypeDef is the same for all tests and consists of two credentials, each with five attributes. While testing the system for "User-Role Assignment" related objectives, the users are assigned credentials from XCredTypeDef, which are compared with the role assignment constraints for allowing the assignment, or otherwise. In Table 4, the test objectives specify a particular aspect of the system that is verified using the corresponding policy files. The $RBAC_p$ corresponding to the test policy files is used to create the structural and behavioral models which are further used to generate the static and dynamic test suites. In order to understand the semantics of the test objectives used in Table 4, we consider two examples of "Role Hierarchy" and "User-Role Assignment." We also tested the system for the handling of incorrect and duplicate files.

5.2.1 Role Hierarchy

Role hierarchy represents the objective to verify the correct implementation of X-GTRBAC hierarchy semantics. This is achieved by generating such policy files in which a single user is assigned to the most senior role from among a total of k roles (see the XURAS column in Table 4). The dynamic test suite is then used to verify a user's ability to activate all the junior roles as per the A-hierarchy semantics. The I-hierarchy semantics are verified by only assigning p permissions to the most junior role, and then verifying the assignment of the same to all senior roles through the static test suite. The user and role activation cardinalities are set to k and 1, respectively. The "Add Cred= true" clause imply that, user has credentials specified in XUS which are compared with role attributes in XURAS for UR assignment. Moreover the "Activ const=false" clause of XRS implies that roles do not have any activation restrictions. This permits the tests to only focus on the verification of hierarchy semantics and not the UR assignment or the role activation semantics. The "DSoD=null" and "SSoD=null" clauses imply that there are no DSoD and SSoD constraints on the UR assignments and activations.

5.2.2 User-Role Assignment

The objective of user-role assignment is to verify the correct implementation of UR assignment controls as specified in the X-GTRBAC system. As already mentioned, the UR assignment is allowed only if the corresponding user credentials satisfy the role assignment constraints specified in XURAS. This is verified by creating XUS and XRS for an equal number (k) of roles and users. All users in XURAS are assigned to all the roles such that the evaluation of assignment constraints would not allow the assignments for k/2 users out of the total k users. In the X-GTRBAC system, the assignment constraints can consist of multiple assignment conditions that are first evaluated individually, and then collectively, through the use of one of the logical operators from the set {OR, NOT, AND}. Moreover, the assignment conditions can also have multiple logical expressions whose collective evaluation is again determined on the basis of the logical operator specified in the assignment condition. The assignment constraints are therefore constructed such that all desired combinations of logical operators for the assignment conditions, expressed as logical expressions, are checked for either true or false value. The XURAS construction in this manner therefore permits the static test suite to fully verify the correctness of the evaluation logic corresponding to the UR assignment constraints. The "User-Role Assignment empty" test is designed to verify the system for the case where although there are users

and permissions in XUS and XPS respectively, but no assignments for same are explicitly made in the system.

5.3 Measuring Test Adequacy

We used statement coverage and condition coverage to assess the adequacy of the tests generated using proposed model-based testing approach. Code coverage values are an indicator of the ability of the tests to exercise various parts of the system. We used the Clover tool [Clover] to measure code coverage in terms of "statement" and "conditional" coverage. For the test parameters given in Table 4, the values used for measuring the code coverage are shown in Table 5.

Table 5: Test Parameters

Test objectives	Parameter Values	
	k	p
User Cardinality	3	2
Role Cardinality	3	2
DSoD	3	2
SSoD	3	2
Role Hierarchy	3	4
User-Role Assignment	20	2
User-Role Assignment Empty	3	2
Role Activation Constraints	20	2

Except for "User-Role Assignment" for which only the static test suite was executed on the X-GTRBAC system, both the dynamic and static test suites were executed on the system for the remaining test objectives. The resultant coverage results for the complete system and individual classes[7] are presented in Tables 6 and 7, respectively. The coverage results are for a total of 29 classes in 21 files with a total of 7,381 lines of code for 367 methods. The coverage results for the both tables are categorized into statement, conditional, and methods coverage. The "initial coverage" in Table 7 shows the coverage results after we first executed the tests on the system by directly using the policy files, generated as per the details in Section 5.2, without any modifications.

Table 6: Coverage Results: Complete System

Coverage	Conditional	Statement	Methods	Total
Initial	86%	94%	91.8%	91.7%
Final	97.2%	97.8%	95.4%	97.4%

The reasons for less than 100% coverage were then investigated and shown in the "Initial Comments" column in Table 7 Based on "Initial Comments," corresponding to a test objective, variants of test policy files were created to test the required functionality. The "final coverage" depicts

[7]The coverage of GUI classes is not measured as the tests directly use the API of the X-GTRBAC system and are not designed to test the GUI classes in the system.

the coverage results obtained after the modified test suites were executed on the system. The "Final Comments" column indicates the reasons of less than perfect coverage even after execution against the modified test suite. In most cases, the reason for less than complete coverage is either the presence of redundant methods, included in the system for future extensions, or the presence of infeasible conditionals.

Table 7: Coverage Results: Individual Classes

Class	Initial Coverage	Initial Comments	Final Coverage	Final Comments
CredType	95.2%	Multiple addition of same credentials not checked	100 %	
CredType.Attribute	100%		100%	
DOMReader	96.4%	Exception not tested for instantiation of DocumentBuilder	96.4%	Same as initial comments
DSDRoleSet	82.6%	One method is redundant, multiple additions of same role not checked, method called for displaying DSoD set not checked	91.3%	One method is redundant
GTRBACModule	93.5%	One method is redundant, exceptions not tested for files not found when trying to close/open the file or parsing the file. One conditional not tested for true	97%	Exceptions for file closing not tested. The conditional cannot be tested for true because decision parameter always has same value. One method is redundant
LogicalExpr (LX)	98%	One conditional not tested which checks for operator to be null	100%	
LX.SimplePredicate	100%		100%	
PermRoleAssign (PRA)	90%	Conditional for checking no permission addition not tested for false	90%	Conditional cannot be tested so because all paths lead to true value
PRA.AssignPermission	76.7%	One method is redundant	76.7%	Same as initial comments
Permission	86.7%	Three methods used for displaying relevant information not tested. One conditional not tested for false	96.7%	The conditional cannot be tested for false because all calls to it can only lead to true value
Policy	69.4%	Nine methods are redundant. Ten methods used for displaying policy sheets not tested. Four conditional not tested for false	91.2%	Four conditionals cannot be tested for false because all calls only lead to true value. Nine methods are redundant
Role	98.1%	Two conditionals to check addition of same DSoD/SSoD not tested for true	100%	
Role.RoleCondition	100%		100%	
Role.RoleConstraint	92.8%	Three logical expression operations not checked for both true and false values	100%	
SSDRoleSet	73.9%	Two methods are redundant, a conditional to check duplicate addition of same role is not checked for true	82.6%	Two methods are redundant
Session	100%		100%	
User	91.5%	One method is redundant; a conditional to check duplicate addition of same credential is not tested for true. Absence of required credential not tested. Conditional for checking absence of logical expression in assign condition not tested for false.	95.8%	Five conditionals cannot be tested for all values because of constraints on all the paths reaching them. One method is redundant.
UserRoleAssign (URA)	92.1%	Two methods are redundant. A conditional to check duplicate addition of same user not checked for true	94.7%	Two methods are redundant.
URA.AssignCondition	100%		100%	
URA.AssignConstraint	93.1%	Conditional checking for no match of opCode not tested for true. Conditional to check change in duration expression not tested for false	100%	
URA.CandidateUser	100%		100%	
XCredTypeDef_DTDScanner	95.9%	Conditionals for matching tags/attribute names in the scanned XML file not tested for false	100%	
XGTRBACMain	100%		100%	
XPRAS_DTDScanner	96%	Conditionals for matching tags/attribute names in the scanned XML file not tested for false	100%	
XPS_DTDScanner	96.8%	″	100%	
XRS_DTDScanner	97.4%	″	100%	
XSoDDef_DTDScanner	96.8%	″	100%	
XURAS_DTDScanner	96.6%	″	100%	
XUS_DTDScanner	98.7%	″	100%	

Tables 6 and 7 show that the proposed model-based testing approach is able to achieve high code coverage. These results are dependent on the correct identification of the test objectives and the selection of the right parameters for the corresponding test policy files.

5.4 Measuring Test Effectiveness

Test adequacy was also measured using program mutation [Demillo78]. Program mutation creates versions of the original program, known as mutants, through simple syntactic changes. The original program and the mutants are then executed against the test cases to assess their adequacy. If the test cases are able to distinguish a mutant from the original program then that mutant is considered *distinguished*. Mutants, other then the ones distinguished, are considered *live*. A mutant could be *live* because of one of two reasons: (a) the test cases are not strong enough to distinguish it from the original program and (b) the program logic does not change from the original in the mutated program i.e. the mutant is semantically equivalent to the original program. The latter type of *live* mutants are considered *equivalent* and in general their identification is an undecidable problem. Test effectiveness is measured as the ratio of distinguished mutants to the total number of non-equivalent mutants. This ratio, multiplied by 100, is also known as the *mutation score* or *mutant score*. Higher mutation score reflects a more effective test set. Program mutation has thus been widely used to compare the effectiveness of different testing strategies [Kim00, Briand04, Andrews05].

Table 8: Method Level Mutation Operators [Mujava]

Operator	Description
AOR	Arithmetic Operator Replacement
AOD	Arithmetic Operator Insertion
AOI	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

We used the Mujava tool [Mujava, Ma05] to perform mutation testing of the XGTRBAC system. Mujava provides a framework for both the efficient generation of mutants and test case execution for programs written in Java. Mujava uses two types of mutation operators, class level [Ma02] and method level [Offutt96]. It uses an extended set of method level mutation operators that

are based on the selective operator set proposed by Offutt et al. [Offutt96]. The method and class level mutation operators are shown in Tables 8 and 9, respectively. Further details of the class level and traditional mutation operators are in [Ma02, Ma05, Offutt96]. Mujava differentiates between *live* and *distinguished* mutants by comparing the output from the execution of the original program against that of a mutant on a test case.

The results of mutation testing on the XGTRBAC system are summarized in Table 10. The mutants were generated for the same set of class files (the inner classes are not shown separately) as in Table 7. No mutant was generated for XGTRBACMain class because it simply initializes the testing framework. Also, unused code, identified during the measurement of code coverage, was not mutated. Mujava generates mutants, compiles them, and executes them automatically against the test cases, provided in a separate class file. The original program and the mutants were executed against the same set of tests used during coverage measurement.

Table 9: Class Level Mutation Operators [Ma05], [Mujava]

Language Feature	Operator	Description
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	overriding method calling position change
	IOR	Overriding method rename
	ISD	super keyword deletion
	ISI	super keyword insertion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	new method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
Java-Specific Features	JTD	this keyword deletion
	JTI	this keyword insertion
	JSI	static modifier insertion
	JSD	static modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

The results in Table 10 show a cumulative mutation score of 94%. A mutation score of 88% was obtained corresponding to class mutants. The lack of 100% mutation score is explained in the following section. Equivalent mutants were identified through manual analysis.

Table 10: Mutation Testing Results (Categorized by Classes)

Class	Method Mutants					Class Mutants				
	Total	Distinguished	Mutant Score (initial)	Equi-valent	Mutant Score (final)	Total	Distinguished	Mutant Score (initial)	Equi-valent	Mutant Score (final)
CredType	30	24	80%	5	96%	14	13	92%	0	92%
DOMReader	0	-	-	-	-	3	1	33%	1	50%
DSDRoleSet	16	10	62%	6	100%	6	3	50%	0	50%
GTRBACModule	218	201	92%	13	98%	6	1	16%	5	100%
LogicalExpr	132	127	96%	5	100%	24	20	83%	4	100%
PermRoleAssign	26	24	92%	2	100%	18	12	66%	0	66%
Permission	43	42	98%	0	98%	35	28	80%	0	80%
Policy	376	362	96%	14	100%	39	20	51%	0	51%
Role	377	338	90%	12	93%	118	115	97%	0	97%
SSDRoleSet	16	10	62%	6	100%	6	3	50%	0	50%
Session	17	13	76%	2	87%	3	3	100%	-	100%
User	246	220	89%	20	97%	28	18	64%	2	69%
UserRoleAssign	139	112	81%	15	90%	29	19	65%	0	65%
XCredTypeDef_DTDScanner	78	62	79%	0	79%	24	13	54%	4	65%
XPRAS_DTDScanner	90	79	87%	0	87%	71	56	78%	13	97%
XPS_DTDScanner	66	62	94%	0	94%	79	65	82%	11	96%
XRS_DTDScanner	287	256	89%	1	90%	222	173	77%	33	92%
XSoDDef_DTDScanner	122	111	90%	0	90%	111	90	81%	17	96%
XURAS_DTDScanner	203	203	100%	-	100%	188	154	81%	26	95%
XUS_DTDScanner	214	189	88%	0	88%	152	108	71%	20	82%
All Classes (cumulative)	2696	2445	91%	101	94%	1176	915	78%	136	88%

6 DISCUSSION

The major task in developer oriented security testing, as is the case with security testing of XGTRBAC system, is the creation of suitable $RBAC_p$ specifications to adequately test the system. This is important because $RBAC_p$ is the sole source for construction of both structural and dynamic models which are in turn used to generate the static and dynamic test suites, respectively. The test suites will only be able to adequately exercise the implementation if the corresponding $RBAC_p$ specifications are able to completely exercise all the rules in the related rule set. The high code coverage results for XGTRBAC system are thus primarily due to the correct identification of the test objectives and selection of associated parameters for the creation of $RBAC_p$ specifications.

The results of mutation testing indicate a strong correlation between code coverage and mutation score for method mutants. This is also expected because method mutants modify the expressions in the program by replacing, adding, or inserting primitive operators; therefore, if tests have good statement and conditional coverage then there reasonable chances of such mutants getting distinguished. As compared to method mutants, mutation score for class mutants is not as high (88% as

compared to 94% for method mutants). The following discussion is based on Table 11 which presents the results of the mutation testing categorized by individual operators. It is to be noted that the operators for which no mutants were generated are not included in Table 11.

Table 11: Mutation Testing Results (Categorized by Operators)

Method Level Mutation Operators						
Operator	Total	Distinguished	Live	Equivalent	Mutant Score	
					Original	After Equivalence Analysis
AOR	178	178	0	0	100%	100%
AOD	29	1	28	21	3%	12.5%
AOI	1489	1329	160	53	89%	93%
ROR	463	411	52	27	89%	94%
COR	8	6	2	0	75%	75%
COI	20	18	2	0	90%	90%
COD	6	5	1	0	83.3%	83.3%
LOI	503	497	6	0	99%	99%
Combined	2696	2445	251	101	91%	94%
Class Level Mutation Operators						
Operator	Total	Distinguished	Live	Equivalent	Mutant Score	
					Original	After Equivalence Analysis
PRV	116	96	20	3	83%	85%
JTI	6	6	0	-	100%	100%
JTD	6	6	0	-	100%	100%
JSI	75	30	45	15	40%	50%
JSD	20	0	20	18	0%	0%
JDC	2	2	0	-	100%	100%
EAM	918	746	172	100	81%	91%
EMM	33	29	4	0	88%	88%
Combined	1176	915	251	136	78%	88%

It can be observed that the mutation score corresponding to all the major contributing method level operators (AOI, LOI, ROR, AOR) is quite high (93%, 99%, 94%, 100%), thus resulting in a overall good mutation score for method mutants. More than half (55%) of the total method level mutants are due to the application of the AOI operator that inserts basic unary and short-cut arithmetic operators in the code [Mujava]. As compared to AOR mutants, for which the mutation score is highest (100%), the score corresponding to AOD mutants is much lower (12.5%) and has the highest ratio of equivalent mutants (75%). Our analysis of AOD mutants revealed that the remaining 7 live mutants, after equivalence analysis, were not distinguished by the test cases because of the limited check on the return value of some function calls.

Application of "EAM: accessor method change" class mutation operator generates most of the class mutants (81% of total class mutants correspond to the application of EAM operator). Consequently most of the non-equivalent live class mutants also belong to EAM mutants (63% of all

live class mutants). EAM operator replaces an accessor method name with that of other compatible accessor methods e.g. EAM_1 live mutant of Permission class changes "for (int i = 0; i < policy.getPRAssignCount(); i++)" to "for (int i = 0; i < policy.getDSDRoleSetCount(); i++)". The test cases are not able to distinguish this change because of the same value of policy.getPRAssignCount() and policy.getDSDRoleSetCount() established through the test policy files. In order to verify that the subject mutant is not really equivalent, we actually created a test case to distinguish it. Our investigation revealed similar reasons for nearly all of the other live EAM mutants.

Although there are only 75 mutants (6% of total class mutants) generated through the application of "JSI: static modifier insertion" class mutation operator, yet 50% of the JSI mutants (30 out of 60) are determined to be live even after equivalence analysis. As the JSI operator adds the static modifier to change instance variables to class variables [Mujava], therefore if only a single instance of an object is created by the test cases, the corresponding JSI mutant cannot be distinguished from the original code. The above reason is the prime cause of the inability of the test cases to distinguish all the live JSI mutants. It is also interesting to observe that the test cases were not able to distinguish any class mutants corresponding to the application of "JSD: static modifier deletion" operator which removes the static modifier to change the class variables to instance variables [Mujava]. The JSD mutants were generated by Mujava only for the document scanner classes (last 7 classes in Table 10) which are used for parsing the policy sheets, so only a single instance of these classes will be always created and hence most of the corresponding JSD mutants could not be distinguished by any test case and were thus considered equivalent.

The above discussion highlights the fact that while performing mutation testing for class mutants, their impact should be carefully considered while creating the $RBAC_p$ specifications. As in our case study, such analysis was not made at the initial stage of test policy construction (Section 5.2), therefore the mutation score for class mutants is not very high.

Mutation testing is based on the premise that the faults injected in the system through the application of mutation operators, are representative of the actual possible faults. The mutation score can therefore be used as a predictor of the fault detection effectiveness of the corresponding testing strategy. A recent study by Andrews et.al. [Andrews05] also concludes that careful application of mutation testing can provide a good estimate of the fault detection capability of the given testing strategy. However, more experiments are needed to generalize these results to the assessment of fault detection effectiveness of security testing strategies.

7 RELATED WORK

7.1 Access Control Policies

Ferraiolo and Kuhn [Ferra92] proposed Role Based Access Control (RBAC). RBAC is shown to be a more flexible approach as compared to DAC and MAC and can be used to represent both DAC and MAC policies [Osborn00]. In DAC, the basic premise is that subjects have ownership over objects of the system and subjects can grant or revoke access rights on the objects they own to other subjects at the original subject's discretion. Subjects can be users, groups, or processes that act on behalf of other subjects. In MAC, the access is governed on the basis of subjects and objects classifications. The subjects and objects are classified based on some predefined sensitivity levels. MAC policy is focused towards controlling information flow with the aim to ensure confidentiality and integrity of information, whereas DAC lacks in providing this support. RBAC has several advantages that allow it to provide simplified security management [Bert99]. These include the abstraction of roles and use of role hierarchy, principles of least privilege and separation of duty (SoD), and policy neutrality [Josh01]. These advantages distinguish RBAC from other models as a powerful model for specifying policies and for specifying rules from any arbitrary organization-specific security model.

7.2 Model Checking based Testing

Model checking [Clark99] has also been used for software testing [Amma98, Gargan99]. *Model checking* is an automatic technique used for verifying finite state systems. The properties to be verified (normally specifications) are usually expressed as formulas in temporal logic. The formal model of the system and the temporal formulas are then fed as input to a model checker which returns "true" if the property holds or generates a counter example. The capability of model checker to generate counter examples is used to create test cases [Amma98]. The mutation analyses of specifications yield mutants which are used by the model checker to generate counter examples. Franseco et.al. [Franc03] have used *model checking* to verify "secure information flow" and "secure termination" properties for programs written in high-level languages. Model checking technique has also been used to analyze security flaws in programs for which control flow graph is available [Besson01]. In [Besson01], a formalism based on a linear-time temporal logic is introduced for specifying global security properties pertaining to the control flow of the program. A model checking based approach for finding security flaws in programs and verifying the absence of certain classes of vulnerabilities in them has been presented in [Chen02]. Dependence of these techniques on the availability of control flow information

restricts their usage for software products for which structural information is not available, which is mostly the case with COTS components.

Model checking has also been used for verification of sequential circuits [Burch94]. A tool for automatic test pattern generation (ATPG) using model checking to detect physical defects in an asynchronous circuit has been suggested in [Marco97]. A method similar to model checking has been proposed for verifying finite state machines by using Ordered Binary Decision Diagrams [Coud89].

7.3 Specification Based Testing

Specification based testing has proposed using model-based specification languages such as Z and VDM. Z and VDM have been used to represent the software specifications formally using mathematical models. A predicate oriented approach based on VDM specifications is presented in [Dick93]. Using Z based specifications for testing is discussed in [Stocks93], [Hier97], and [Burt2000]. Specification based testing has also been investigated by converting the informal specifications to *cause effect graphs* and applying a *Boolean operator* strategy to design and select test cases [Parad97]. A method to generate tests from Boolean specifications of software is given in [Weyu94]. In this study the quality of test cases is determined by applying the test cases against few mutation-style faults. A scenario-based object oriented testing framework, which uses test scenario specification as input has been proposed in [Tsai03] for adaptive and rapid testing.

7.4 Security Testing

A model-based approach for security functional testing has been proposed in [Chan04]. The text based specifications of security functions are first transformed to SCR (software cost reduction) formal language specifications which are later used to create the SCR behavioral model. The behavior model is used to create test vectors which are then executed on the product. The necessity of risk-based approaches for security testing is argued in [Potter04]. In risk-based approach, the risks are first identified in the system and tests are then created based on the identified risks. A white box testing approach for vulnerability testing of software systems using a variant of the fault injection technique has been proposed in [Du00]. In this approach, each environment perturbation is considered a fault which is then injected into the system and the resultant system response is observed to determine the system's ability to tolerate the fault. Another approach for penetration/vulnerability testing recommends construction of tests based on perceived risks and the integration of test results back into the organization software development life cycle [Arkin05].

8 SUMMARY AND CONCLUSIONS

We have proposed a model-based approach for security testing of access control systems. The proposed approach generates the static and dynamic test suites, for testing an implementation, from the structural and behavioral models respectively of the corresponding RBAC policy specification. It thus provides a systematic procedure for the construction of test suites.

Code coverage and mutation score are measured in a case study to assess test adequacy and effectiveness of the proposed model-based security testing approach. The results from our case study indicate that in addition to the test generation technique, test adequacy and effectiveness is also dependent on the correct identification of suitable RBAC specifications, which essentially acts as the source for the proposed structural and dynamic models. The results of test coverage and effectiveness measurements from our case study indicate a strong correlation between mutation score for method mutants and code coverage. Moreover, it was observed that the mutation score of class mutants is also strongly dependent on the construction of RBAC specifications. In the future, we would like to extend our model-based testing approach to access control systems using RBAC policy specifications with temporal constraints.

References

[Ahmed03] T. Ahmed and A.R. Tripathi, Static Verification of Security Requirements in Role Based CSCW Systems. *Proc.of ACM SACMAT,* pp. 196-203, 2003.

[Ahn2000] G. Ahn, R. Sandhu. Role-Based Authorization Constraints Specification. *ACM Transactions on Information and System Security*, 3(4), pp. 207-226, 2000.

[Amma98] P. E. Ammann, P. E. Black, and W. Majurski. Using Model Checking to Generate Tests from Specifications. *Proc. of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pp. 46-54, 1998.

[Andrews05] J.H. Andrews, L.C. Briand and Y. Labiche. Is mutation an appropriate tool for testing experiments?. *Proc 27th international conference on Software engineering (ICSE'05)*, pp. 402-411, 2005.

[Arkin05] B. Arkin, S. Stender, G. McGraw. Software penetration testing. *IEEE Security & Privacy Magazine*, 3(1), pp. 84 – 87, 2005.

[Beiz90] B. Beizer. Software Testing Techniques. *Van Nostrand Reinhold*, 1983.

[Beiz95] B. Beizer. Black Box Testing. *John Wiley & Sons, Inc*. 1995

[Bert99] E. Bertino, E. Ferrari and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1), pp. 65-104, 1999.

[Besson01] F. Besson, J. Jensen, D.L. Me´tayer, and T. Thorn. Model Checking Security Properties of Control Flow Graphs. *Journal Computer Security*, 9(3), pp. 217-250, 2001.

[Bhatti05] R. Bhatti, J.B.D. Joshi, E. Bertino and A. Ghafoor. X-GTRBAC: An XML-based Policy Specification Framework and Architecture for Enterprise-wide Access Control. *ACM Transactions on Information and System Security (TISSEC)*, 8(2), pp. 187-227, 2005.

[Binder99] R.V Binder. Testing Object-Oriented Systems - Models, Patterns, and Tools. *Object Technology, Addison-Wesley*, 1999.

[Briand04] L.C. Briand, M. Di Penta and Y. Labiche. Assessing and improving state-based class testing: a series of experiments. *IEEE Transactions on Software Engineering*, 30(11), pp. 770- 783, 2004.

[Burch94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 13(4) pp. 401 – 424, 1994.

[Burt2000] S. Burton. Automated Testing From Z Specifications. *TR YCS-2000-329 Department of Computer Science ,University of York, Heslington, York*.

[Chan04] R. Chandramouli. M. Blackburn. Automated Testing of Security Functions using a combined Model & Interface driven Approach. Proc. 3*7th Hawaii International Conference on System Sciences*, pp. 299-308, 2004

[Chen02] H. Chen, D. Wagner. MOPS: An infrastructure for examining security properties of software. *Proc. 9th ACM Conference on Computer and Communications Security,* pp. 235-244, 2002.

[Chow78] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3), pp. 178-187, 1978.

[Clark99] E. Clarke, O. Grumberg, and D. Peled. Model Checking. *MIT Press*, 1999.

[Clover] Java Coverage Tool. www.cenqua.com/clover/

[Cohen96] D.M. Cohen, S. R. Dalal, J. Parelius and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software.* 13(5), pp. 83-88, 1996.

[Coud89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. *International Workshop on Automatic Verification Methods for Finite State Systems*. *Lecture Notes in Computer Science Springer Verlag,* 407,1989.

[Dalal98] S.R. Dalal, A. Jain, N. Karunanithi, J. M Leaton and C. M. Lott. Model-based testing of a highly programmable System. *Proc. Ninth International Symposium on Software Reliability Engineering (ISSRE)*, pp. 174-179, 1998.

[Demillo78] R.A. DeMillo, R.J. Lipton and F.G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11(4). pp. 34-41, 1978.

[Demillo87] R.A. DeMillo, M. McCracken, R.J. Martin and J F. Passafiume. Software testing and Evaluation. *The Benjamin/Cummins Publishing Company*. 1987

[Dick93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. *Proc. FME '93: Industrial-Strength Formal Methods, Springer-Verlag Lecture Notes in Computer Science*, 670, pp. 268-284, 1993.

[Dima99] A.Dima, J. Wack and S. Wakid. Raising the bar on software security testing. *IT Professional,* 1(3), pp. 27-32, 1999.

[Du00] Wenliang Du and A. P. Mathur. Testing for Software Vulnerability Using Environment Perturbation. *Proc. workshop On Dependability Versus Malicious Faults, International Conference on Dependable Systems and Networks (DSN 2000)*, pp. 603-612, 2000.

[Ferra92] D. Ferraiolo and R. Kuhn. A role-based access control. *Proc. of the NIST-NSA National (USA) Computer Security Conference*, pp. 554-563, 1992.

[Ferra01] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security*, 4(3), pp. 224–274, 2001.

[Fried02] G. Friedman, A. Hartman, K. Nagin and T. Shiran. Projected state machine coverage for software testing. *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 134-143, 2002.

[Fuji91] S. Fujiwara , G.v. Bochmann, F. Khendek, M. Amalou and A. Ghedamsi. Test selection based on finite state models . IEEE Transactions on Software Engineering, 17(6), pp. 591 – 603, 1991.

[Gargan99] A. Gargantini, C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Proc. of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering,* pp. 146 – 162, 1999.

[Gon70] G. Gonenc. A method for the design of fault detection experiments. *IEEE transactions on computers*. Vol C-19. pp. 551-558, 1970.

[Good75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *Proc. Int Conf Reliable Software*, 1975.

[Hier97] R. Hierons. Testing from a Z specification. *Software Testing Verification and Reliability*, 7(1), pp. 19–33, 1997.

[Ip96] C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Formal methods in system design*, pp. 41-75, 1996.

[Jilani98] L.L. Jilani and A. Mili. Estimating COTS integration: An analytical approach. *Proc. 5th Maghrebian Conf. on Software Eng. and Artificial Intelligence*, Dec. 1998.

[Jiwn02] K. Jiwnani and M. Zelkowitz . Maintaining Software with a Security Perspective. *Proc. International Conference on Software Maintenance (ICSM'02)*, pp. 194-203, 2002.

[Josh01] J.B.D. Joshi, W.G. Aref, A. Ghafoor and E.H. Spafford. Security Models for Web-based Applications. *Communications of the ACM*, 44(2), pp. 38-72, 2001.

[Kim00] S. Kim, J.A. Clark, and J.A. McDermid. Investigating the Effectiveness of Object-Oriented Testing Strategies with the Mutation Method. *Software Testing, Verification & Reliability,* 11(3), pp. 207-225, 2001.

[Lee94] D. Lee and M. Yannakakis. Testing finite-state machines: state identification and verification. *IEEE Transactions on computers*, 43(3), pp. 306-320, 1994.

[Lupu99] E. Lupu and M. Sloman. Conflicts in Policy-based Distributed Systems Management, IEEE *Transactions on Software engineering,* 5(6), pp. 852-869, 1999.

[Ma02] Y.S. Ma, Y.R. Kwon and J. Offutt. Inter-Class Mutation Operators for Java. *Proc 13th International Symposium on Software Reliability Engineering*, pp. 352-363, 2002.

[Ma05] Y.S. Ma, J. Offutt and Y.R. Kwon. MuJava: An Automated Class Mutation System. *Journal of Software Testing, Verification and Reliability*, 15(2), pp. 97-133, 2005.

[Marco97] A.P. Marco, E. Pastor and J.Cortadella. Symbolic Techniques for the Automatic Test Pattern Generation for Speed-Independent Circuits. *Tech. Report Num. RR-1997-04, Universitat Politècnica de Catalunya.* 1997.

[Mujava] Mujava, A mutation system for Java programs. *http://salmosa.kaist.ac.kr/LAB/MuJava/*

[Offutt96] J. Offutt, A. Lee, G. Rothermel, R. Untch and C. Zapf. An Experimental Determination of Sufficient Mutation Operators. *ACM Transactions on Software Engineering Methodology*, 5(2), pp. 99-118, 1996.

[Offutt03] J. Offutt, S. Liu, A. Abdurazik and P. Ammann. Generating Test Data From State-based Specifications. *The Journal of Software Testing, Verification and Reliability*, 13(1), pp. 25-53, 2003.

[Osborn00] S. L. 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*, 3(2), pp. 85-106, 2000.

[Parad97] A. Paradkar, K. C. Tai and M. A. Vouk. Specification-based testing using cause-effect graphs. *Annals of Software Engineering,* 4, pp. 133 – 157, 1997.

[Potter04] B. Potter and G. McGraw. Software security testing. *IEEE Security & Privacy Magazine,* 2(5), pp. 81 – 85, 2004.

[Sabn88] K.K. Sabhani and A.T. Dahbura. A Protocol Test Generation Procedure. *Computer Networks*, 15, pp. 285-297, 1988.

[Sandhu94] R. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communications*, 32(9), pp. 40-48, 1994.

[Sandhu98] R. Sandhu. Role activation hierarchies. *Proc. third ACM workshop on Role-based access control*, pp. 33-40, 1998.

[Stock96] P. Stocks and D. Carrington. A Framework for Specification-Based Testing. *IEEE transactions on software engineering*, 22(11), pp. 777-793, 1996.

[Spence94] I. Spence and C. Meudec. Generation of Software Tests from Specifications. *International Conference on software quality management*, pp. 517-530, 1994.

[Stocks93] P. A. Stocks. Applying formal methods to software testing. *PhD Thesis. Department of Computer Science, The University of Queensland, Australia*. 1993.

[Thom03] H.H Thompson. Why security testing is hard? *IEEE Security & Privacy Magazine*, 1(4), pp. 83-86, 2003.

[Tsai03] W.T. Tsai, A. Saimi, L. Yu and R. Paul. Scenario-based object-oriented testing framework. *Proc. third International Conference on Quality Software,* pp. 410-477, 2003.

[Weyu94] E. Weyuker, T. Goradia and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5), pp. 353–363, 1994.