CERIAS Tech Report 2006-24

SCALABLE AND EFFECTIVE TEST GENERATION FOR ACCESS CONTROL SYSTEMS THAT EMPLOY RBAC POLICIES

by Ammar Masood, Arif Ghafoor and Aditya Mathur

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

Scalable and Effective Test Generation for Access Control Systems that Employ RBAC Policies

Ammar Masood* Arif Ghafoor† Aditya Mathur‡

August 3, 2006

Abstract

Representation of Role Based Access Control (RBAC) policies as finite state models and three conformance testing procedures for generating tests from these models are proposed. A test suite generated using one of the three procedures has excellent fault detection ability but is astronomically large. Two approaches to reduce the size of the generated test suite were investigated. One is based on a set of six heuristics and the other directly generates a test suite from the finite state model using random selection of paths in the policy model. A fault model specific to the implementations of RBAC systems was used to evaluate the fault detection effectiveness of the generated test suites; the model incorporates both mutation-based and malicious faults. Empirical studies revealed that adequacy assessment of test suites using faults that correspond to first-order mutations may lead to a false sense of confidence in the correctness of policy implementation. The second approach to test suite generation is most effective in the detection of both first-order mutation and malicious faults and generates a significantly smaller test suite than the one generated directly from the finite state models.

Keywords: Role Based Access Control (RBAC), Finite state models, State explosion, W-method Wp-method, Fault model, First-order Mutation faults, and Malicious faults.

1 Introduction

Access control is essential for safe and secure access to software and hardware resources. An access control implementation is responsible for granting or denying authorizations after the identity of a requesting user has been validated through an appropriate authentication mechanism. Operating systems, database systems, and other applications employ policies to constrain access to application functionality, file systems, and data. Often these policies are implemented in software that serves as a front end guard to the protected resources or is interwoven with the application. It is important that the access control software is correct in that it faithfully implements a policy it is intended to. Hereafter an implementation of access control policies is referred to as ACUT (for Access Control Under Test).

A number of reported common vulnerabilities and exposures [26] are related to design and/or coding flaws in access control modules of an application. Testing remains indispensable despite advances in the

*Purdue University, Electrical and Computer Engineering; West Lafayette, IN 47907. ammar@ece.purdue.edu.

†Purdue University, Electrical and Computer Engineering; West Lafayette, IN 47907. ghafoor@ecn.purdue.edu. Corresponding Author.

‡Purdue University, Computer Science; West Lafayette, IN 47907. apm@purdue.edu.

formal verification of secure systems [1, 32, 25] and in static or dynamic program-analysis based techniques [12, 30] because verification only guarantees correctness of the design under certain assumptions. Any faults in the implementation due to, for example, coding errors, incorrect configuration, and hidden or "backdoor" functionality could jeopardize the effectiveness of corresponding (access control) specification [44]. It therefore becomes critical to assure that the underlying implementation conforms to the desired policy, and hence testing becomes essential. Given a safe and consistent policy P currently in effect, we ask: *What tests, when successful, would ensure that ACUT enforces P and no other policy ?*

To answer the above question the use of a "complete Finite State Machine (FSM)" based conformance testing strategy was investigated. The strategy investigated was to construct an FSM model of the RBAC (role based access control) policy [19, 41, 42] and then generate tests from the model using the well known W-method [9]. Evaluations of tests generated using the W-method often use Chow's FSM-based fault model [9, 23, 38]. We examined the fault coverage of the complete FSM based conformance testing technique with respect to an RBAC fault model that consists of faults derived from first order mutations. The proposed technique provides complete fault detection with respect to the RBAC fault model that can be mapped to Chow's fault model as shown in Section 6.1.

The fault coverage of complete FSM strategy is further assessed by extending the RBAC fault model and thus considering the non-mutation faults (referred as *malicious faults*). It is determined that tests so generated are able to detect a particular class of malicious faults. The complete fault coverage for malicious faults can only be achieved if white box coverage measures are used for test enhancement. In the absence of ACUT code, usage of black box based technique cannot provide any guarantees about its effectiveness in detecting malicious faults. Hence we suggest using white box coverage criteria such as data flow and mutation, to facilitate enhancement of FSM generated tests for providing complete coverage of malicious faults. Certainly, code reviews [5] and inspection may also assist in detecting such faults.

The number of states in a complete FSM model of the expected behavior of access control enforcement logic can easily reach 3^{50} in an application with ten users and five distinct roles. While complete FSM based conformance testing technique turns out to be highly effective in detecting RBAC faults, the size of the finite state model and that of the generated test suite is astronomical thereby rendering it unsuitable for practical use. Thus we ask: *How does one scale down a test suite generated from a finite state model without "significant" degradation in the fault detection effectiveness of the scaled down test suite ?*

To answer the above question we investigated two approaches. The first approach referred to as "heuristics based strategy," uses one or more heuristics to reduce the state space of the FSM through state abstractions. Such abstractions are also used in formal verification techniques [1, 25]. These heuristics were derived from a knowledge of the structure of access control policies. While the heuristics did lead to a drastic reduction in the size of the model, they also resulted in reduced fault detection effectiveness; whether or not this reduction is "significant" is subjective and can only be quantified in specific instances of implementation. The second approach referred as "constrained random test selection" strategy (CRTS), reduces the size of the test suite by random selection of paths of fixed length from the complete FSM. Given a sufficient number of tests, the CTRS strategy is likely to have better fault detection than "heuristics based strategy" because the abstractions used by the heuristics only consider a local view of the implementation in contrast to the complete view considered by the CRTS strategy that randomly selects paths from an FSM.

The complete FSM based, the heuristics based, and the CRTS strategies discussed previously target conformance testing of the ACUT with respect to a single RBAC policy. To guarantee that ACUT will correctly enforce all policies, it is essential to perform functional testing across a representative set of policies. We propose a functional testing technique based on the use of any of the proposed complete FSM, heuristics,

and CRTS based conformance testing procedures to perform functional testing of ACUT by using multiple policies.

We conducted an empirical evaluation to assess the cost, effectiveness, and the cost-benefit ratio associated with the usage of the three procedures in functional testing of a prototype RBAC system. The effectiveness was measured using first-order mutation faults [16] and manually injected malicious faults. Usage of both the mutation faults and malicious faults in effectiveness measurement provided insight into the inability of state abstraction based heuristic procedure to provide complete fault coverage for malicious faults despite being adequate with respect to mutation faults.

Contributions: (a) A technique for modeling the expected behavior of access control systems (ACUT) that enforce RBAC policies. (b) A fault model for RBAC system based on selective mutations and on (possibly) malicious code. (c) Conformance testing strategy based on complete FSM, heuristics and CRTS procedures, (d) A technique for functional testing of ACUT and (e) evaluation of the usage of the three conformance testing procedures in the proposed functional testing technique through a case study.

Organization: RBAC and Chow's FSM based test generation method are reviewed in Section 2. The testing context of proposed test generation method is described in Section 3. Section 4 describes how to construct a finite state model from a given RBAC policy. Section 5 discusses the conformance relation used as the basis for conformance testing procedures. A fault model used for assessing the effectiveness of the tests generated is described in Section 6. The proposed conformance test generation procedures are described in Section 7. Section 8 describes in detail the proposed functional testing technique. Section 9 reports an empirical study conducted to assess the cost and fault detection effectiveness associated with usage of proposed conformance testing procedures in functional testing of a system. Related work is reviewed in Section 10. Section 11 summarizes the proposed approach.

2 Background

2.1 Role Based Access Control

RBAC is often used to protect resources from unauthorized access. For example, a bank has a set of roles, e.g. Teller and Customer, users, e.g. John and Mary, and resources, e.g. accounts. Authorized personnel in this bank assign users to roles. In turn each role has associated permissions that allow a user to perform tasks. For example, a Teller is allowed to deposit a check into a customer's account and a customer may not have the permission to transfer money from her account to another customer's account.

An RBAC policy P is a 16-tuple $(U, R, Pr, UR, PR, \leq_A, \leq_I, I, S_u, D_u, S_r, D_r, SSoD, DSoD, S_s, D_s)$, where

- U and R are, respectively, finite sets of users and roles,
- Pr is a set of permissions,
- $UR \subseteq U \times R$ is a set of allowable user-role assignments,
- $PR \subseteq Pr \times R$ is a set of allowable permission-role assignments,
- $\leq_A \subseteq R \times R$ and $\leq_I \subseteq R \times R$ are, respectively, activation and inheritance hierarchy relations on roles,
- $I = \{AS, DS, AC, DC, AP, DP\}$ is a finite set of allowable input requests for the ACUT, where AS, DS, AC, DC, AP, DP are, respectively *Assign*, *Deassign*, *Activate*, and *Deactivate* requests for user-role assignment and activation and *Assign* and *Deassign* for permissions-role assignments.

- $S_u, D_u : U \rightarrow Z^+$ are, respectively, static and dynamic cardinality constraints on U, where Z^+ denotes the set of non-negative integers.
- $S_r, D_r : R \rightarrow Z^+$ are, respectively, static and dynamic cardinality constraints on R.
- $SSoD, DSoD \subseteq 2^R$ are, respectively, static and dynamic Separation of Duty (SoD) sets
- $S_s : SSoD \rightarrow Z^+$ specifies the cardinality of the $SSoD$ sets
- $D_s : DSoD \rightarrow Z^+$ specifies the cardinality of the $DSoD$ sets

We explicitly attach the policy P with each element of the above 16-tuple when there is a need to distinguish it from that of another policy. For example $UR(P)$ and $UR(P')$ are the UR assignments corresponding, respectively, to policies P and P'.

The activation hierarchy relation (A-hierarchy) $r_i \leq_A r_j$ implies that a user u_k assigned to r_j is also able to activate r_i without being assigned to it i.e. $(u_k, r_i) \notin UR$ ([41]). The inheritance hierarchy relation (I-hierarchy) $r_i \leq_I r_j$ means that a permission p_k assigned to r_i is also accessible by r_j without being actually assigned to it i.e. $(p_k, r_j) \notin PR$ ([41]). The static (dynamic) cardinality of a user specifies the maximum number of roles it can be assigned to (can activate). Similarly, the static (dynamic) cardinality of each role specifies the maximum number of users who can be assigned to (can activate) this role.

The $SSoD$ ($DSoD$) [2] specifies the sets of roles to which users can only be simultaneously assigned (can simultaneously activate) provided such assignments (activations) do not violate the $SSoD$ ($DSoD$) set cardinality constraint i.e. $S_s(SSoD)$ ($D_s(DSoD)$). $S_s(SSoD)$ ($D_s(DSoD)$) constrains the maximum number of roles to which a user can be simultaneously assigned (can simultaneously activate) in the given $SSoD$ ($DSoD$) set.

Note that while $(u, r) \in UR$ implies that assignment of u to r is allowable, u is authorized for assignment only when (i) an input request $AS(u, r)$ is received and (ii) the static user and role cardinality and SoD constraints are satisfied at the time the assignment request is received. For user u to be authorized to activate role r, (i) input request $AC(u, r)$ must be received, (ii) u must be assigned to r or permitted via $\leq_A$, and (iii) dynamic user and role cardinality and SoD constraints must be satisfied.

The above definition is adequate to illustrate the proposed test generation strategy. However, we are aware that variations of the above policy exist such as [19] [28] and [2]. In the NIST RBAC [19] a user is required to initiate a session in order to activate a subset of his assigned roles. A session maps a user to possibly many roles and a one to many mapping exists between a user and his sessions. The finite state model of a given RBAC policy, described in Section 4, can be extended to model the user sessions. Three types of control flow dependency constraints have been considered in [28] and a number of variants of the $SSoD$ and $DSoD$ relations have been considered in [2] . These constraints can be represented in the finite state model of the given policy. The RBAC variant considered in this work is closely related to the non-temporal version of the X-GTRBAC system [6]. A sample policy follows.

Example 1. Consider the following policy P with two users, one role, and two permissions.

$$U = \{u_1, u_2\}, R = \{r_1\}, Pr = \{p_1, p_2\},$$

$$UR = \{(u_1, r_1), (u_2, r_1)\}, PR = \{(p_1, r_1), (p_2, r_1)\},$$

$$S_u(u_1) = S_u(u_2) = D_u(u_1) = D_u(u_2) = 1, S_r(r_1) = 2, D_r(r_1) = 1, \leq_A = \leq_I = \{\ \}$$

Each permission p_1 and p_2 in Pr is associated with functions and resources related to the application under consideration. ■

2.2 Test generation from finite state machine

A Mealy finite state machine (FSM) $M = (Q, q_0, X, Y, \delta, O)$, where Q is a finite set of states, $q_0 \in Q$ a unique initial state, X and Y, respectively, the input and output alphabets, $\delta : Q \times X \rightarrow Q$ the state transition function, and $O : Q \times X \rightarrow Y$ the output function.

Let $s = x_1 x_2 \ldots x_{k-1} x_k$, $x_i \in X, 1 \leq i \leq k$, be a string of length $k > 0$ over the input alphabet X, also written as $s \in X^+$. We write $O^+(q_m, s) = r$ for string r of length k over the output alphabet Y, when $O(q_{l_j}, s_j) = r_j, 1 \leq j \leq k$, $\delta(q_{l_j}, s_j) = q_{l_{j+1}}$, $q_m = q_{l_1}$, where all q's are states in Q. Similarly, we write $\delta^+(q_m, s) = q_k$, when $\delta(q_{l_j}, s_j) = q_{l_{j+1}}, 1 \leq j \leq k$, $q_m = q_{l_1}, q_k = q_{l_k}$. States $q_i, q_j \in Q, i \neq j$, are considered distinguishable by string $s \in X^+$ if $O^+(q_i, s) \neq O^+(q_j, s)$.

Let $W = \{w_1, w_2, \ldots, w_m\}$ be a finite set of non-empty strings. W is a characterization set for M if for any two states $q_i, q_j \in Q, i \neq j$, there exists a $w \in W$ such that $O^+(q_i, w) \neq O^+(q_j, w)$. W is considered minimal if for all sets W', $|W'| < |W|$, W' is not a characterization set for M.

M is complete when δ and O are defined for all states in Q. M is connected when each state in Q is reachable from q_0. M is minimal when there exists a characterization set for M.

Let M be a minimal, complete, and connected FSM that models the expected behavior of the ACUT. Let $n = |Q|$. Let m an estimate of the number of states in FSM M' that models actual behavior of the ACUT. Note that the testing methodology proposed here does not require an explicit formulation of M'. Concatenation of sets R and S is written as $R.S = \{uv | u \in R, v \in S\}$. The W-method for generating test suite T from M proceeds in the following steps [9, 24].

1. Construct the testing tree Tr.
2. Construct the transition cover set P_t from Tr. P_t contains all paths in Tr from its root to all internal nodes and leaves.
3. Find the state characterization set W.
4. Test set T is constructed as follows.

$$T = \begin{cases} \cup_{i=0}^{i=m-n} P_t.X^i.W & \text{if } m - n > 0 \\ P_t.W & \text{otherwise} \end{cases}$$

Several other methods are available for generating a test suite from an FSM [3, 40]. However, the W-method has the best and proven fault detection ability [9, 38, 43]. The procedure given above can be easily modified to generate a test suite using the Wp method [23]. The resultant test suite is smaller than that generated by the W-method and retains the fault detection effectiveness.

Chow's fault model for finite state machines, consists of four faults: operation, transfer, extra state, and missing state. An operation fault occurs when M' has a transition with an incorrect input or output label. A transfer fault occurs when M' has a transition that terminates at an incorrect state. Chow has shown that given m, a correct estimate of the number of states in M', T is guaranteed to detect all faults in the fault model. Empirical studies have shown that the fault coverage is high even when m is incorrect.

Step 3 in the test generation procedure above is not needed when the states in the ACUT are observable. In this case there is a significant reduction in the size of the test suite as the formula in Step 4 reduces to:

$$T = \begin{cases} \cup_{i=0}^{i=m-n} P_t.X^i & \text{if } m - n > 0 \\ P_t & \text{otherwise} \end{cases}$$

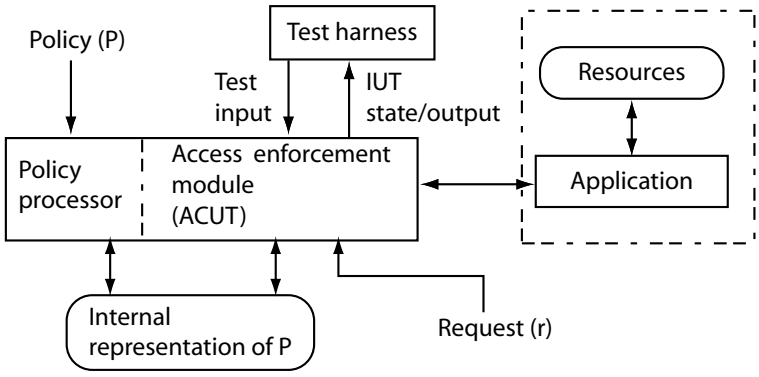

Figure 1: Interaction between an application, access control enforcement module (ACUT), and the protected resources. Test harness contains test cases generated using a finite state model. Test cases are to test the policy enforcement mechanism, not the application.

To enable the observation of states, the ACUT may need addition of output statements at appropriate locations. Each output statement sends a unique state identifier to the test harness described in Section 3. This output allows the harness, also serving as an oracle, to check if the ACUT has arrived at the correct state after processing an input request.

A further reduction in the size of T becomes possible when the ACUT states are observable. In such cases there is no need to record paths from the root of Tr to each internal node as state identification is not required. Thus P_t now consists only of paths from the root of Tr to each leaf.

3 Testing context

Figure 1 shows the context of the applicability of the proposed test generation approach. As shown, the access enforcement module is the system under test (ACUT). Prior to testing, the ACUT is initialized with a policy P. It is assumed that a Policy processor performs this initialization task. The Policy processor constructs an internal representation of P for subsequent access by the ACUT. Often the internal representation is a table containing various relations described in Section 2.1. This division of tasks between two submodules allows for flexibility in the specification of policies. For example, P could be an XML file [6] or it could be specified using a GUI [29] attached to the Policy processor.

A request received by the ACUT is authenticated against the policy and, if *granted* passed to the Application. While the ACUT is shown as a separate module, it could also be interfaced with the application in other ways. For example, the incoming request could first enter the Application and then passed to the ACUT for authentication.

The Test Harness in Figure 1 encapsulates the generated test cases. Each test case t could assume one of two forms: (r, q) or (r, rp), where $r = r_1, r_2, \ldots, r_{k-1}, r_k$ is a sequence of $k > 0$ requests that belong to the input alphabet I, $q = q_1 q_2, \ldots, q_{k-1}, q_k$ is the expected state transition sequence, and $rp = rp_1, rp_2, \ldots, rp_k$ is the expected response sequence.

Each request is parameterized with appropriate inputs. For example, an Assign request $AS(u, r)$ specifies user u and a role r. The (r, q) form is selected to test ACUT where state transitions are observable. The (r, rp) form is used when state transitions are not observable but response to each input request is. For each

input request we use subscripts i and j to denote, respectively, user u_i and role r_j. For example AS_{ij} is an abbreviation for "Assign u_1 to role r_j".

Testing begins with the ACUT in its initial state. Test t is applied by sending each request in t to the ACUT. The corresponding state transitions are observed and compared against the expected state transitions in q. The behavior of the ACUT is assumed to conform to the expected behavior as per policy P when the observed state sequence is identical to q. The ACUT is brought to its initial state prior to the application of the next test input.

In cases when the ACUT does not return its current state information it is reasonable to assume that the *granted* and *denied* actions in response to each request are observable. However, in this case our test generation strategy uses the state characterization set W [9] in the generation of the test cases.

The context described above, and the test generation strategy proposed herein, allows testing the ACUT for a single policy. In general, it is important to test the ACUT for a variety of policies to ensure that it will indeed correctly enforce access for any given policy. The functional test generation technique described in Section 8 uses multiple policies to test an ACUT.

4 Modeling the expected behavior of the ACUT

The first step in the proposed test generation strategy is to model the expected behavior of the ACUT as an FSM M. Following assumptions are made to model an RBAC policy: $X = \{AS, DS, AC, DC\} \subset I$ and $Y = \{$*granted*, *denied*$\}$, where AS, AC, DS, DC are, respectively, abbreviations for requests to assign a user to a role, activate a user-role pair, deassign a user from a role, and deactivate a user-role pair. *granted* and *denied* denote two possible responses of an ACUT to any of the four request types. Unless stated otherwise, M is considered complete, i.e. for every input there is a transition from every state, and connected, i.e. every state is reachable from the initial state. The permission-role assignments are ignored to simplify the presentation.

A state in M is a sequence of pairs of bits, one pair for each user-role combination as in the table below. For example, given two users u_1 and u_2, and one role r_1, a state is represented as a pattern of two consecutive pairs of bits. In this case, 1011 indicates that u_1 is assigned to role r_1 but has not activated r_1 and u_2 is assigned to r_1 and has activated it.

Pattern	**Role**	
	Assigned	**Activated**
00	No	No
10	Yes	No
11	Yes	Yes
01	Not used	Not used

FSM(P) refers to an FSM that models the expected behavior of an ACUT with respect to policy P. Figure 2(a) shows an FSM model M that represents the behavior of the ACUT required to implement policy P in Example 1. It consists of eight states corresponding to the different assignments and activations in effect. In general, for u users and r roles, the upper bound on the number of states in the FSM corresponding to a policy is 3^{ur}. In Section 7.2 we propose heuristics to reduce the size of the model and hence that of the test set.

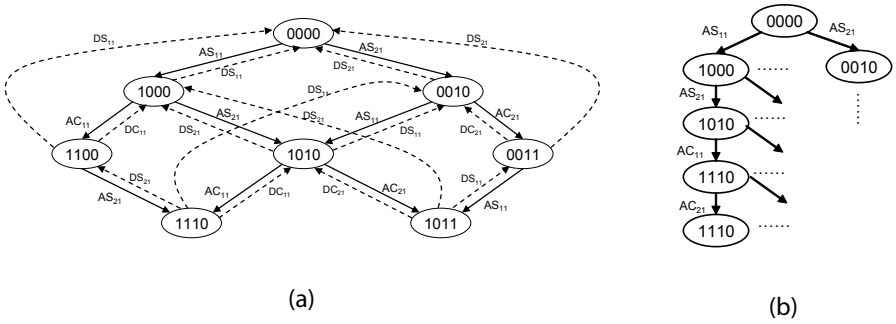

Figure 2: (a) A complete finite state behavioral model derived from the RBAC policy in Example 1 and (b) its partial testing tree. Expected response is not shown. For each transition between two states, the response is *granted*. Self-loops corresponding to *denied* response are not shown to keep the figure uncluttered.

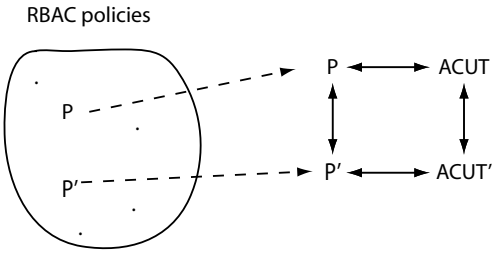

Figure 3: Policy and implementation conformance. An arrow, in either direction, is to be read as "conforms to."

5 Conformance Relation

Let $\mathcal{R}$ denote the set of all RBAC policies, X an organization that uses role based access control to protect its resources, and ACUT′ an implementation used by X to enforce any RBAC policy over some duration. Given the definition of an RBAC policy, $\mathcal{R}$ is infinitely large. It is reasonable to assume that, in any given duration, X enforces one policy $P \in \mathcal{R}$.

Now suppose that ACUT is an implementation that correctly enforces P. However, a faulty ACUT′ might enforce $P' \in \mathcal{R}$, where P' is not the same as P. The goal of conformance testing for access control is to ensure that ACUT′ is free of faults that may lead to incorrect enforcement of P. The proposed fault model is derived with this goal in view. Figure 3 illustrates the proposed conformance relationship amongst policies and their respective implementations.

Let P be an RBAC policy in effect and ACUT a correct implementation that enforces P and no other policy. Let $UR_a \subseteq U \times R$, $UR_c \subseteq U \times R$, and $PR_a \subseteq P \times R$ be sets of, respectively, current user-role assignments, user-role activations, and permission-role assignments with respect to P. Let $Rq(up, r), up \in (U \cup Pr)$, be a well formed request such that $Rq \in I$ and $(up, r) \in (U \times R)$ for $up \in U$, and $(up, r) \in (Pr \times R)$ for $up \in Pr$. $Rq(up, r)$ is considered ill-formed when any one or more of the following conditions does not hold: $Rq \in I$, $up \in (U \cup Pr)$, and $r \in R$.

The status S of an ACUT is the set $\{UR_a, UR_c, PR_a\}$. Each of the three marked subsets of S is empty at the start of ACUT execution, and hence $S = \{\{\}, \{\}, \{\}\}$. S changes in response to requests $Rq(up, r) \in I$ and policy P. We write $S'_{\text{ACUT}} = S_{\text{ACUT}}[Rq(up, r)]$ to indicate that the updated status of ACUT in response to request Rq is S'_{ACUT} if the status prior to receiving $Rq(up, r)$ was S_{ACUT}.

ACUT′, an implementation under test, is said to conform *behaviorally* to ACUT with respect to policy P, under the following conditions.

1. For all requests $Rq(u, r) \in I$, if $S'_{\text{ACUT}} = S_{\text{ACUT}}[R(up, r)]$ then $S'_{\text{ACUT}'} = S'_{\text{ACUT}} = S_{\text{ACUT}'}[Rq(up, r)]$.

2. For all ill-formed requests $Rq(up, r)$, $S_{\text{ACUT}}[Rq(up, r)]$ and $S_{\text{ACUT}'}[Rq(up, r)]$ remain unchanged.

Stated informally, behavioral equivalence implies that ACUT′ (a) assigns (deassigns) and activates (deactivates) a role only if such assignment (deassignment) and activation (deactivation) is allowable by the current policy in effect, (b) assigns (deassigns) a set of permissions to (from) a role only if allowable by the current policy in effect, and (c) ignores ill-formed requests. It is important to note that ACUT′ correctly enforces only the policy currently in effect and no other policy. Certainly this is not to be interpreted in the sense that ACUT′ is capable of enforcing only one policy. In fact an implementation of RBAC policies is expected to enforce any RBAC policy. However, at any instant in time it only enforces the policy that is in effect. We assume the existence of a mechanism to change the currently effective policy.

6 RBAC Fault model

Policies P and P' are considered conforming when ACUT and ACUT′ are behaviorally conformant. The aim of conformance testing of ACUT′ is to establish the behavioral conformance between the ACUT′ and ACUT. Conformance of ACUT′ with respect to the ACUT can also be intuitively thought of as absence of any faults in the ACUT′ i.e. faults in P'. Given a policy $P \in \mathcal{R}$, where $\mathcal{R}$ is infinitely large and is the set of all possible RBAC policies, $\mathcal{R}$ can be partitioned into two subsets; set of conforming ($\mathcal{R}^P_{conf}$) and faulty ($\mathcal{R}^P_{fault}$) policies with respect to P. The conformance testing of ACUT′ thus implies verifying that P' does not belong to the set of faulty policies i.e. $P' \notin \mathcal{R}^P_{fault}$.

As $\mathcal{R}^P_{fault}$ can be infinite therefore devising a test strategy that guarantees detection of all types of faults, i.e. guaranteeing that $P' \notin \mathcal{R}^P_{fault}$ can be impractical. Except possibly through exhaustive testing, it is impossible to show through testing that ACUT′ is behaviorally equivalent to ACUT. Traditionally in conformance testing of systems the total number of possible implementations is restricted to a finite set by assuming a fault model for the implementation [38]. The fault model depends on the specification model and it specifies the types of faults that can be encountered in an implementation.

It is important to note that faults are directly related to the conformance relation, in this case behavioral conformance, used in between the implementation and the specification [7]. Fault model can thus be used for fault coverage assessment of a testing technique [9, 31, 38]. We used mutation based approach, similar to the one widely used for deriving fault models for FSM based specifications [38], to construct the fault model of RBAC by considering all the mutants of P. The proposed fault model is based on behavioral conformance. It is to assist in the evaluation of the fault detection effectiveness of tests generated for testing ACUT′.

The RBAC fault model restricts $\mathcal{R}^P_{fault}$ to be finite by only considering such $P' = (U, R, Pr, UR', PR', \leq'_A, \leq'_I, I, S'_u, D'_u, S'_r, D'_r, SSoD', DSoD', S'_s, D'_s) \in \mathcal{R}^P_{fault}$ that can be derived from $P = (U, R, Pr, UR, PR, \leq_A, \leq_I, I, S_u, D_u, S_r, D_r, SSoD, DSoD, S_s, D_s)$ by making simple changes to P. Note that all $P' \in \mathcal{R}^P_{fault}$ have the same sets of users, roles, permissions and inputs and these sets are equivalent to the corresponding sets in P.

The set $\mathcal{R}^P_{fault}$ is obtained by recursively applying the set mutation operators to the sets $UR, PR, \leq_A, \leq_I, SSoD$ and $DSoD$ in P and element modification operators to the range of functions S_u, D_u, S_r, D_r,

Table 1: RBAC faults due to mutations of elements of P.

Structures Mutated	Possible Impact on ACUT′ (Fault)
$UR, S_u, S_r, SSoD, S_s$	UR1, UR2
$PR, \leq_I$	PR1, PR2
$\leq_A, D_u, D_r, DSoD, D_s$	UA1, UA2

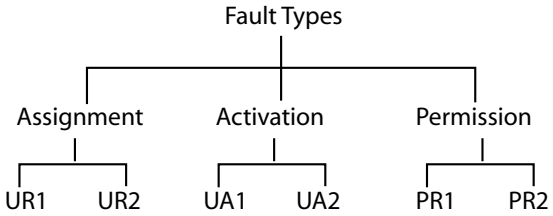

Figure 4: A fault model for evaluating the effectiveness of tests of RBAC implementations.

S_s and D_s. We consider three types of set mutation operators; modification of an element, addition of an element and removal of an element. The semantics of element modification depends on the type of the element, which in case of another set implies recursive application of set mutation operators on the element. The application of modification operator to an integer z in the range of function $F \in (S_u, D_u, S_r, D_r, S_s, D_s)$ would change the value to $z + 1$ and $z - 1$.

For an element $(u, r) \in UR$, the effect of modification operator could be in three ways: exchange of u with another $u' \in U, u' \neq u$, exchange of r with another $r' \in R, r' \neq r$, and exchange of both u and r. The impact of modification operator on an element $(p, r) \in PR$ would also be similar. For a role pair $(r_i, r_j) \in \leq_A$, the modification could cause replacement of either one role r_i or r_j with $r' \neq r_i, r_j$ or of both roles with $(r', r'') \neq (r_i, r_j)$. The modification would have similar effect on a role pair $(r_i, r_j) \in \leq_I$. The set mutation operators will be recursively applied on the $SSoD$ and $DSoD$ sets. Considering the individual element r_i of a set $(r_i, r_j, \ldots, r_k) \in SSoD$, the modification operator would result into exchange of r_i with $r' \neq r_i, r' \in R$.

Table 1 illustrates that the application of above mentioned mutation operators on the elements of P would result into a policy P' which implies possible presence of various faults in the ACUT′. As observed from Table 1, the RBAC fault model consists of three types of faults: user-role assignment, user-role activation, and permission-role assignment. As shown in Figure 4, each fault is further categorized into two subcategories. Fault type UR1 restricts an authorized user from being assigned to a role or leads to an unauthorized deassignment. Fault type UR2 may lead to unauthorized role assignments. PR1 faults restrict a permission being assigned to an authorized role or cause an unauthorized deassignment. PR2 faults assign a permission to an unauthorized role. UA1 and UA2 faults are similar to UR1 and UR2 and impact role activation.

The proposed fault model is complete in that that any violation of an RBAC policy corresponds to at least one of the six fault types. For example suppose that role r_1 can be activated by at most one user at any instant. Now, suppose that while u_1 has activated r_1, a request for activation of r_1 by u_2 is allowed. This is an instance of the UA2 fault. Examples of other fault types can be constructed similarly.

6.1 Relation between FSM and RBAC Fault Model

The table below shows the correspondence between the RBAC fault model in Figure 4 and the one proposed by Chow [9] for FSM. This correspondence is important in that it allows us to argue that tests generated from FSM(P), as explained later in Section 7.1, are able to achieve complete fault coverage of the RBAC faults. This correspondence can be easily established through comparison between FSM(P'), where P' is the mutated policy, and the FSM(P). As an example a P' obtained from P by adding a $(u, r) \notin UR(P)$ pair to the $UR(P')$, thus causing a UR2 fault, would lead to at least one extra state fault in the FSM(P').

RBAC fault model	**FSM fault model**[9]
UR1, UA1, PR1	Transfer fault, Missing state fault, Output fault
UR2, UA2, PR2	Extra state fault, Output fault, Transfer fault

Example 2. Figure 5 relates UA2 and UR1 faults to the corresponding faults in an FSM. Figure 5(b) illustrates a UA2 fault. It shows an extra state and a transfer fault when the ACUT correctly denies an activation request for u_1 but moves to an incorrect state. Figure 5(c) illustrates a UR1 fault due to incorrect transfer and output faults causing deassignment of an authorized user.

Figure 5(d) illustrates a case where the state of the ACUT is correct though an incorrect output is generated. While we do not consider this as a fault, the test harness in Figure 1 will be able to detect such an inconsistency in the ACUT. ■

It is easy to establish the correctness of fault correspondence between faults in the enforcement of a policy and those in an FSM using Figure 5 as a guide.

6.2 Malicious faults

Faults that cannot be modeled as first order simple or higher order mutations of an RBAC policy are placed in the malicious faults category. While a competent programmer makes programming mistakes that could often be considered as combination of one or more simple mutations [16], a malicious programmer may inject faults that simulate devious ways of tricking an ACUT into malicious behavior. A malicious fault leads an ACUT into a malicious state only under certain conditions. If such faults remain undetected during testing, then during operation they offer a loophole to a malicious user.

Certainly, some malicious faults could be modeled using first or higher order mutations, those that cannot be modeled are of particular interest. *Note that we are not implying that malicious faults cannot be detected*

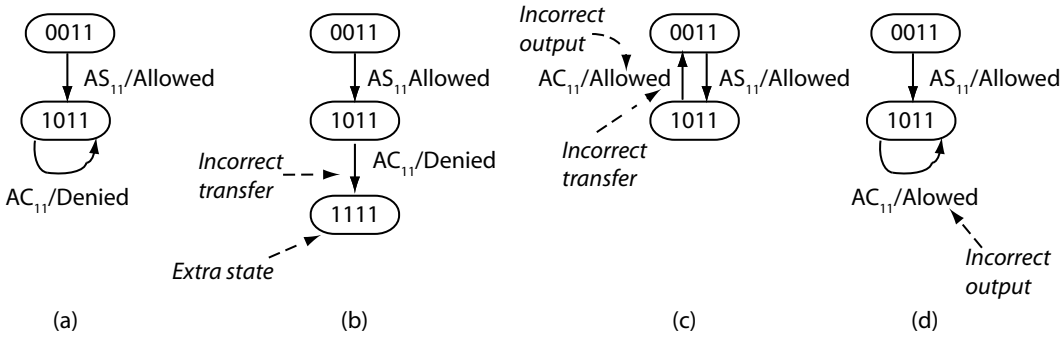

Figure 5: Mapping of the UA2 and UR1 faults to those in Chow's fault model. (a) Correct transitions extracted from Figure 2. (b) Extra state and transfer fault. (c) Transfer and output faults. (d) Output fault that does not correspond to the proposed fault model because the ACUT remains in a consistent state (1011).

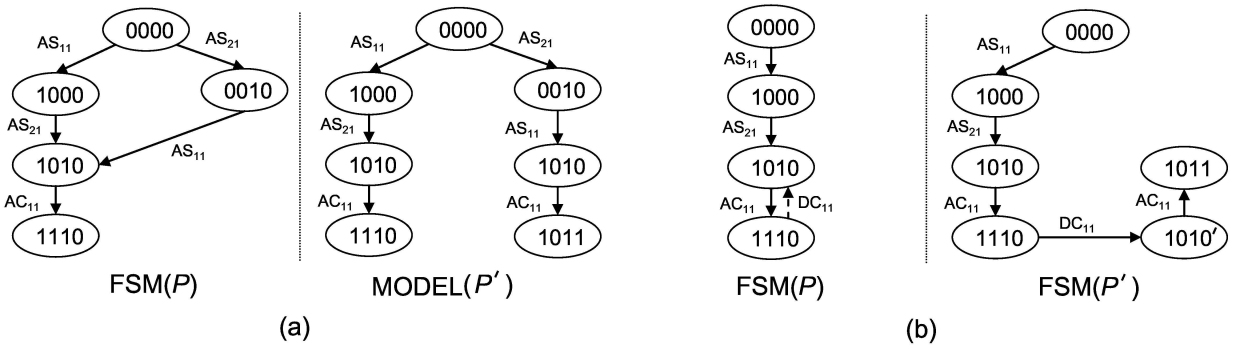

Figure 6: (a) Example of a sequence-based fault, (b) Mapping of counter-based fault to Chow's fault model

by a test adequate with respect to mutation or other white-box adequacy criteria. Rather, we are interested in investigating the effectiveness of practically viable black-box test generation techniques in detecting such faults.

Counter-based faults: A counter-based fault is said to exist in an ACUT if it contains reachable code that incorrectly grants, denies, or aborts a request based on counts of events. Consider an ACUT that counts the number of specific user-role pair activations. If this count is greater than some integer $k > 0$ then the ACUT behaves in a devious way and not otherwise. This is an example of a counter-based fault. One could construct a variety of examples of such counter-based faults.

I/O-based faults: An I/O-based fault is said to exist in an ACUT if it has reachable code that incorrectly grants, denies, or aborts a malformed request. For example, consider an ACUT that allows a user with a special user ID not in the set of authorized users in an RBAC policy to be assigned to a role and then activate the role. This is an example of an I/O-based fault that violates an RBAC policy. A test generated using an automata theoretic approach would not have any request of the kind $AS(sid, r_1)$ for some special ID sid and a known role r_1. As another example, an ACUT might process a special request, e.g. "deassign all users" that does not belong to the input set I of RBAC. Other examples of I/O-based faults can be constructed easily by adding elements that do not belong to sets U, R, and Pr. We ignore faults based on the output alphabet as these can be detected easily by a test harness.

While code-coverage based test enhancement techniques, e.g. branch coverage, will likely force the construction of a test case that reveals the presence of malicious code mentioned in the two examples above, this may not be possible when using tests generated from an FSM model using automata theoretic technique such as the W method. Also, this might not be possible when the ACUT is a unit of a large application, no unit testing is performed, and there is no requirement to cover all branches or other elements of the code.

Sequence-based faults: A programmer could also inject one or more malicious sequences into the code. For example, an ACUT might allow an invalid access when a specific sequence of user-role assignments have been activated. Note that here it is not the count but a specific sequence of user-role assignment or activation inputs that leads the ACUT into a malicious state. While such faults are malicious, any such sequence constitutes a path in the FSM model and hence can be detected by at least one test generated using the automata theoretic method.

It is to be noted that malicious faults are also categorized as per the fault classification given in Figure 4. Sequence-based faults cannot be related with FSM faults as they are the result of non-FSM behavior of the ACUT. An example of sequence-based fault corresponding to FSM of Figure 2 is given in Figure 6(a).

The input sequences AS_{21}, AC_{11} and AS_{11}, AC_{11} given in the states (1000) and (0010) respectively, of FSM(P) would always lead to the state (1110), whereas the later sequence would lead to the state (1011) in the FSM(P'). Counter-based faults can be related with extra state FSM fault, as illustrated in Figure 6(b). The input sequence $AC_{11}, DC_{11}, AC_{11}$ given in state (1010) of FSM(P) would always lead to the state (1110), whereas same sequence applied in state (1010) of FSM(P') would lead to state (1011). I/O based faults cannot be related to FSM faults.

7 Generation of Conformance Test Suite

In this section we propose three procedures, with varying cost and fault detection effectiveness, for generating the test suite for conformance testing of the ACUT with respect to a specific RBAC policy. The cost is measured while assuming that states are observable and hence W set is not considered in the test generation (Section 2.2). The upper bound on cost is measured in terms of the total number of state variable queries performed in the execution of a test suite. Cost effectively depends upon the total number of tests, their lengths and the number of state variables. The fault detection effectiveness of these procedures is evaluated with respect to the faults in the RBAC fault model proposed in Section 6.

7.1 Procedure A: Complete FSM based

In this procedure tests are generated from the complete FSM ($M = $ FSM(P)), derived from the policy P, as per the steps outlined in Section 2.2. The complexity of this procedure not only depends on the size of M but also on the observability of states in the ACUT. For the FSM(P) of Example 1 given in Figure 2(a), the test set is partially shown in Figure 2(b). The testing tree is derived using the algorithm given in[9]. It includes at least one path from the initial state to all other states in the FSM(P). The upper bound on the number of states in M and of the test execution cost of this procedure is given in Table 3. For the RBAC policy of Example 1 the empirical comparison between the cost of Procedure A and the cost of heuristics based Procedure B, discussed next, is also made on the basis of the number of tests generated by each procedure, as given in Table 3.

We have already shown (Section 4) how to construct an FSM that captures the behavior of an ACUT that enforces an RBAC policy. The mapping between the simple faults in the proposed fault model and FSM fault model, used by Chow [9], has also been established in Section 6.1. The following claim is based on the above observations; additional supporting arguments are found in [9].

Claim: The proposed method of generating the behavior model and tests therefrom, guarantees a test set that detects all simple faults in the ACUT that correspond to the proposed RBAC fault model when the number of states in the ACUT is correctly estimated.

For malicious faults, it can be observed that Procedure A is unable to detect I/O-based faults, rather all the procedures are unable to detect these. The counter-based faults can be detected by this procedure if the total number of states in the ACUT is accurately estimated. The sequence-based faults are always detected as the test tree contains at least one test case for each path in the FSM.

Table 2 lists a sample of tests generated by assuming state observability and the faults each test case is able to detect in a faulty ACUT. The test generation procedure becomes more complex if state are not observable, though it can be automated. For the FSM in Figure 2, the W-set consists of one sequence of length six: $AC_{11}AC_{21}DS_{11}AS_{11}AC_{11}DS_{21}$. With some effort one could verify that this sequence is adequate to distinguish any two of the eight states in Figure 2.

Table 2: A sample of test inputs obtained from the FSM in Figure 2. In all cases, we assume that the input sequence is applied with the ACUT in its initial state, i.e. 0000.

Test input	**Fault detected**
$AS_{11}, AC_{11}, AS_{21}$	A transfer fault in state 1100 leading to self loop on AS_{21} input would result in a UR1 fault where u_2 is not assigned to r_1 in a faulty ACUT.
AS_{21}	A transfer fault in state 0000 which on input AS_{21} leads to transition to state 1000 instead of 0010 leads to both UR1 and UR2 faults.
$AS_{11}, AS_{21}, AC_{11}$	A transfer fault in state 1010 leading to self loop on input AC_{11} would result in a UA1 fault where u_1 is unable to activate r_1 in a faulty ACUT.
$AS_{11}, AS_{21}, AC_{11}, AC_{21}$	An extra state fault in transition from state 1110 to 1111 would lead to a UA2 fault where u_2 can activate r_1 despite $D_r(r_1) = 1$.

Procedure A provides very good fault coverage but at the expense of very high cost as the size of FSM(P) increases exponentially with the increase in the number of state variables. The next two procedures are aimed at reducing the size of the conformance test suite. The reduction in test suite size can adversely effect the fault detection effectiveness as discussed in the description of these procedures.

7.2 Procedure B: Heuristics based

In this procedure six heuristics, labeled H1 through H6, are used to reduce the size of the model and of the test set. These heuristics are similar to the concept of state abstractions as used in various verification techniques, some of which are discussed later in Section 10. Each heuristic is explained with respect to Example 1 and the column labeled "Complete FSM" in Table 3.

H1: Separating assignment and activation: Construct M_{AS} and $M_{AC_1}, M_{AC_2}, \ldots, M_{AC_k}$, $k > 0$. Here M_{AS} is an FSM that models all assignment requests. For each state $q_i \in M_{AS}$, there is an activation FSM M_{AC_i} that models the expected activation behavior under the assumption that the assignment state remains q_i. Figure 7(a) and (b) show, respectively, M_{AS} and $M_{AC_{11}}$ for the policy in Example 1.

Note that a state in M_{AS} corresponds to assignments whereas that in M_{AC} corresponds to activations. For the model in Figure 2, application of this heuristic leads to an increase in the total number of states from 8 to 12. However, as in Table 3, the reduction in the number of tests is more than double. The reduction is due to the separation of the FSMs.

H2: FSM for activation and single test sequence for assignment: Construct model M_{AC} for activation requests with respect to a single state q_{max} that has the maximum number of assignments in M_{AS}. The single test sequence is the concatenation of requests along a path from the initial state of M_{AS} to q_{max}. For $q_{max} = 11$, $M_{AC} = M_{AC_{11}}$ as in Figure 7(a). Assignments and deassignments are covered using a sequence of AS and DS requests. The number of test cases now reduces from 92 to 11 in the best case.

H3: Single test sequence for assignment and activation: Use one test sequence that includes assignment, activation, de-activation, and deassignment requests for all the user-role pairs in any order. In this case the behavior of the ACUT is tested using a mix of all four types of requests. Doing so reduces the number of tests from 92 to 1 in the best case.

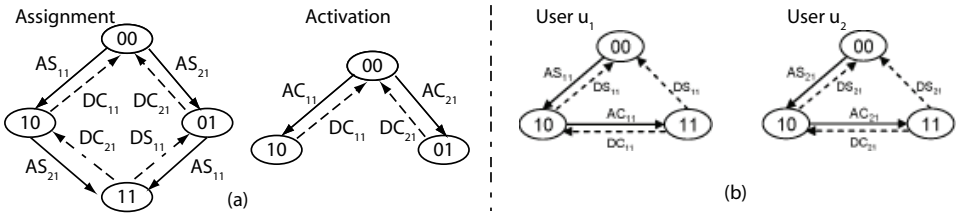

Figure 7: Models constructed by applying (a) H1 and (b) H4. As in Figure 2, self loops and outputs are not shown. Note that for H1 there are four activation FSMs though only $M_{AC_{11}}$ is shown here.

H4: FSM for each user: Construct M_u for each $u \in U$. Apply the test generation procedure separately to each model. Figures 7(b) show M_{u_1} for user u_1 and M_{u_2} for user u_2 for the policy in Example 1. The reduction in the number of states is from 8 to 6 and that in the size of the test set from 92 to 20.

H5: FSM for each role: Construct M_r for each $r \in R$. Apply the test generation procedure separately to each model. As the policy in Example 1 contains only one role r_1, M_{r_1} is the same as in Figure 2. Hence there is no reduction in the size of the model or the test set. However, a reduction is expected when $|R| > 1$.

H6: Grouping users: Create user groups $UG = \{UG_1, UG_2, \ldots, UG_k\}, k > 0$, such that $\cup_{i=1}^{k} UG_i = U$ and $UG_i \cap UG_j = \emptyset, 1 \leq i, j \leq k, i \neq j$. The groups can be created using one or more common attributes, e.g. all users that can be assigned to the same set of roles. The FSM is now constructed assuming that the user field in each input request corresponds to a user group and not to a user. For example, $AS(u_2, r_3)$ is a request to assign a user from UG_2 to role r_3.

Under H6 the meaning of a "state" of an FSM needs reinterpretation. For example, given $k = 2$, "1011" means that a representative from UG_1 is assigned to r_1 and a representative from UG_2 is assigned to, and has activated, r_1. The FSM does not have any notion of a "user" this has been replaced by that of a "representative" of a user group. This heuristic could lead to a significant reduction in the number of states and transitions given the fact that a group assigned to a role cannot be reassigned to that role before it is deassigned. Similar reduction occurs as the same group cannot reactivate a role until it has deactivated it. Note H6 can be combined with the other heuristics above.

When using H6, the test harness randomly selects a *representative* u' for the user group UG upon receiving the first assignment request for UG. In all subsequent tests, it is this representative that is input to the ACUT. u' is deassigned when a DS request for the corresponding group is successful. Selection of a representative for a user group leads to a significant reduction in the number of tests generated due to a reduction in the number of states in the model. In Example 1, the number of tests generated reduces from 92 to 10.

In addition to the six heuristics in Table 3, one could also relax the FSM completeness assumption while generating tests. In one case we do not consider the AS and AC requests in assigned and active states. In another case, we do not consider DS and DC requests in unassigned and inactive states. The last two columns in Table 3 show, respectively, the number of tests generated when the completeness assumption is relaxed.

7.2.1 Impact on fault detection effectiveness

In these heuristics model reduction is achieved by only considering a localized view of the system behavior as compared to the global view maintained by complete FSM. It is obvious that scaling down the model by applying H1 through H6 might have a negative impact on the fault detection effectiveness of the tests

Table 3: Sizes of test sets obtained by applying various heuristics. $|X|$ denotes the number of elements in set X. $T = |U| \times |R|, T_g = |UG| \times |R|$

Heuristic	**Upper bound on**		$\|Tset\|$ **for Example 1**		
	$\|Q\|$	Cost	**Complete FSM**	**Ignore AS, AC in assigned and active states**	**Ignore DS, DC in unassigned and inactive states**
None	3^T	$2T(2T+1)(4T)^{2T+1}$	92	64	40
H1	2^T+2^{2T}	$T(T+1)(2T)^{T+1}(2^T+1)$	44	32	16
H2	2^T	$T(T+1)(2T)^{T+1}+2T$	11	9	5
H3	No FSM	$4T$	1	1	1
H4	$\|U\|3^{\|R\|}$	$2T\|U\|(2\|R\|+1)(4\|R\|)^{2\|R\|+1}$	20	14	8
H5	$\|R\|3^{\|U\|}$	$2T\|R\|(2\|U\|+1)(4\|U\|)^{2\|U\|+1}$	92	64	40
H6	3^{T_g}	$2T_g(2T_g+1)(4T_g)^{2T_g+1}$	10	7	4

generated. Quantifying this impact for arbitrary implementations is possible only in specific instances of an implementation through empirical studies. Here we briefly review the impact due to each heuristic, considering their application to FSM of Example 1 given in Figure 2.

It is possible for an ACUT to behave such that its response to activation requests is dependent on the specific assignment sequence used to arrive at a state. For example, with reference to Figure 7(a), the implementation corresponding to $M_{AC_{11}}$ might behave correctly in response to the request sequence $AS_{21} \rightarrow AS_{11}$ but not with respect to $AS_{11} \rightarrow AS_{21}$. Such non-FSM like behavior of the ACUT could cause faults to remain undetected when H1 is used.

Tests derived using H2, H3, H4, and H5 might miss faults located along certain paths of the complete FSM. For example, heuristic H3 covers only one path along the complete FSM. Thus faults along other paths might remain undetected.

When using H4, H5, or H6, incorrect implementation of cardinality constraints as well as other incorrect assignments and activations, might go undetected. For example, if the number of users is 1000, and $S_r(r_1) = 750$, a faulty ACUT might allow (a) F1: r_1 to be assigned to more than 750 users or (b) F2: allow at most one user to be assigned to r_1. When using H6, fault F2 will be detected by the tests generated from the FSM, but not F1.

When using H6, stress testing can be used to detect F1 and similar faults related to cardinality constraints. The stress points are derived from the cardinality constraints. In the example above, we need to stress the ACUT so that it is asked to assign at least 751 users to role r_1. Certainly, there are several different sequences by which an ACUT can be stressed and that depend on the context of the assignments. Such variations in tests could be covered using several randomly generated request sequences aimed at detecting cardinality and sequence errors that correspond to the faults in the proposed fault model. While such fault are not guaranteed to be revealed by stress and random tests, at least we hope that the chances of their detection are improved.

7.3 Procedure C: The CRTS strategy

The CRTS strategy is aimed at reducing the number of test sequences without requiring reduction in the model size. We use the term "constrained" to describe CRTS to stress the fact that the tests are generated randomly but are constrained by the model. As already mentioned, in contrast to the complete FSM considered by Procedure A, the abstractions used by the heuristics only consider a local view of the system. The

CRTS strategy is precisely aimed at achieving the goal of a reduction in the size of the test suite without incuring the loss of reduced system view as is the case for heuristics used in Procedure B.

The strategy is to first construct pools RTi, RT$j \ldots$, RTt of random tests, where all pools have equal number of tests i.e. $l = |\text{RT}i| = |\text{RT}j| \ldots = |\text{RT}t|$. All tests in a pool RT$i$ have same length i. The lengths $i, j \ldots, t$ of tests in the CRTS pools are selected to be close to or significantly higher then the length of longest test generated by using Procedure A. The value of total number of tests l in a CRTS test pool can be selected based on the comparison with number of tests in the test suites generated using heuristics in Procedure C. A test case in the CRTS pool RTi is constructed through random selection of a path of fixed length i from FSM(P) as test.

Five tests suites are then constructed from each CRTS pool RT$' \in \{\text{RT}i, \text{RT}j \ldots, \text{RT}t\}$ by randomly selecting fixed number p of tests from RT$'$. Given an FSM(P), one could use some statistical criterion, such as the rate of ACUT failure, or an economic criterion such as the maximum number of test sequences, as a criterion to determine the total number of tests p in each CRTS test suite. The finer details of CRTS strategy are easier to understand by considering a practical example as illustrated in the case study in Section 9.2.3.

It is important to note that the CRTS strategy is applied to the non-reduced FSM model as in Procedure A. While the original FSM is astronomically large, CRTS strategy can be easily coded without the need to actually represent the model in internal memory. Procedure C has fixed cost evaluated as the product of total number of tests in a suite, the number of state variables in FSM(P) and the length of tests in that suite.

The fault detection effectiveness of CRTS strategy is expected to be better than Procedure B as the reduced FSM, considered in heuristics, fails to consider a holistic view of the ACUT behavior. Counter-based faults can be detected by the CRTS strategy depending on the count of the events required for the fault to be exhibited and the length of test cases in a CRTS suite. By virtue of random selection of paths, chances of detection of sequence-based faults are also higher than heuristics based strategy. As already mentioned, the effectiveness of CRTS strategy does depend on the length of tests in a test suite, as tests of length comparable with the tests generated from complete FSM are expected to better exploit the ACUT functionality. The results of the case study reported in Section 9 also support these assertions.

8 Functional Testing of ACUT

Conformance testing of ACUT only establishes its conformance with respect to a specific RBAC policy P. Functional testing is required to ensure that ACUT will correctly enforce all policies. Recall that the set $\mathcal{R}$ of all RBAC policies is infinite, thus functional testing implies guaranteeing ACUT conformance with respect to all $P \in \mathcal{R}$. As exhaustive testing is not a viable option, therefore only finite number of policies have to be used for test generation with the intent to fully exploit the ACUT functionality. By restricting the space of policies for which ACUT is tested, it is possible that some parts of the code may not be executed by the generated tests; therefore, some white box adequacy criteria such as mutation and code coverage is required to be used to establish correctness of ACUT functionality. Note that the importance of using white-box adequacy criteria has been stressed by several researchers [16, 45].

8.1 Proposed Functional Testing Methodology

The functional test suite for an ACUT is a pair $< Pset, Tset >$, where $Pset = \{P_1, P_2, \ldots, P_k\}$, $k > 0$ is a finite set of organizational policies and $Tset = \{T_1, T_2, \ldots, T_k\}$ is a finite set of set of test suites, where each T_i is generated from $P_i, 1 \leq i \leq k$. We refer to $Pset$ as a meta test set as it derives $Tset$

that contains test suites that in turn contain test cases. It is to be noted that a $T \in Tset$ can be generated by using either Procedure A, B or C discussed in Section 7. The technique for functional testing of ACUT proceeds in following steps.

1 Generate initial policy set $Pset = \{P_1, P_2, \ldots, P_k\}$

2 Generate $Tset = \{T_1, T_2, \ldots, T_k\}$

3 Test ACUT against each $T \in Tset$, remove any faults discovered

4 Evaluate $Tset$ using one or more white-box criteria such as MC/DC, coverage and mutation

5 Add a new policy P' to $Pset$ if criteria not satisfied. Go to step 3

The feedback loop in step 5 terminates when the adequacy criteria is satisfied.

8.2 How to generate policies ?

The initial policy set is generated manually. There is no set of rules one could use to construct an initial $Pset$ and to decide how many policies should $Pset$ be initialized with. The initial policy set should be as comprehensive as possible for it to cover a large portion of the ACUT code. This implies that collectively policies should contain at least one instance of each constraint specified by RBAC policy definition (Section 2.1).

In the simplest case the policies can be generated according to the organization's needs for resource protection. One could begin with a single policy that is to be implemented to protect a set of resources. e.g. data in an organizational database. Additional policies are then added based on the results of adequacy criteria in step 5. Policies can also be constructed by traditional techniques of equivalence partitioning, boundary value analysis [36], and combinatorial design [13]. For example, one policy might have no DSoD constraints, another one such constraint, and a third one two such constraints. Combinations of SSoD and DSoD constraints can also be used to create additional policies.

When program mutation is used as an adequacy criteria then mutants can be used in constructing the initial policy set, as is the approach (Section 9.2.2) used in the case study discussed next. Program mutation creates versions of the original program, known as mutants, through simple syntactic changes. Some of the mutants could be semantically equivalent to the original program and are thus classified as *equivalent* mutants.

Other than the equivalent mutants all others can be related to the RBAC faults with respect to some policy P. In order to understand this approach consider $F = \{f_1, f_2 \ldots, f_n\}$ as set of all non-equivalent mutants. Initially policy P_1 is constructed based on organization access control requirements and is added to the $Pset$. As F is the set of non-equivalent mutants therefore a subset $F' \subset F$ of these faults would correspond to some RBAC fault with respect to P_1, unless P_1 is trivial and does not exercise any constraints. Faults in the set $F'' = F - F'$ are then analyzed to construct more policies which are then added to the $Pset$. The $Pset$ is considered complete when all faults in F can be correlated with the RBAC faults with respect to at least one policy $P \in Pset$.

9 Empirical evaluation

An empirical study was conducted to assess the cost, fault detection effectiveness, and cost-benefit ratio associated with the usage of the three procedures, described in Section 7, in functional testing of an ACUT.

As already mentioned (Section 7) the cost is measured in terms of the total number of state variable queries performed in the execution of a test suite. Program mutation [16] and manual injection of malicious faults were used to measure the fault detection effectiveness. The cost-benefit ratio is the ratio of the cost of the tests generated using a procedure to the number of faults detected by that procedure. The study was based on an implementation of a a generic access control mechanism named X-GTRBAC [6]. Brief description of X-GTRBAC and details of the study follow.

9.1 The X-GTRBAC system

The X-GTRBAC is a generic authentication and access control application. It can serve as a front-end to any application that needs to authenticate users and enforce access controls based on an RBAC policy. X-GTRBAC is written in Java and the access control portion consists of two sets of modules: a policy initializer and a policy enforcer. The initializer takes as input an RBAC policy coded in XML, checks it for syntactic correctness, and saves the policy in a tabular format. The enforcer accepts requests for user-role activations and deactivations, checks these against the policy constraints, and either allows or denies an incoming request.

Only the policy enforcement subsystem of X-GTRBAC was the target of the case study. This subsystem consists of seven classes listed in Table 4. Only activate and deactivate requests are accepted by the policy enforcer. Assignment of users to roles is done by the policy initializer and hence user-role assignment and deassignment requests are not dynamically accepted by the enforcer.

9.2 Case study: Method and Results

The case study followed the steps for functional testing of ACUT as described in Section 8.1. The steps and results obtained follow.

9.2.1 Prepare X-GTRBAC

We used program mutation as adequacy criteria and thus, as described earlier in Section 8.2, $Pset$ was obtained using analysis of the non-equivalent mutants. Section 9.2.2 describes in detail the application of this approach in the case study. Two types of faults are injected into the policy enforcement module: first order mutations [16], hereafter referred to as simple faults, and sequence-based malicious faults (Section 6.2). The fault detection effectiveness of each procedure is studied with respect to both simple and malicious faults. The set of program mutants is referred as F.

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 for simple faults is measured as the ratio of distinguished mutants to the total number of non-equivalent mutants and the test effectiveness for malicious faults is measured as the ratio of detected malicious faults to total number of malicious faults.The policy enforcer module is mutated by applying all

Table 4: Classes and their characteristics in the policy enforcement subsystem of X-GTRBAC and faults injected.

Class	Method count	LOC mutated	Mutants		Mutant classification			
			Total	Equivalent	UR1	UR2	UA1	UA2
DSDRoleSet	3	15	20	8	0	0	4	8
GTRBAC module	5	95	82	20	21	0	31	10
Policy	9	121	113	14	63	30	1	5
Role	8	140	150	12	21	26	27	64
Session	3	27	9	4	0	0	2	3
SSDRoleSet	3	15	18	6	4	8	0	0
User	5	54	23	2	13	0	4	4

Java mutation operators to the classes listed in Table 4. Mutation operators as defined in the muJava system are used [33]. These include the five traditional operators and twenty three Java class related operators.

Not all methods in a class were subjected to mutation. The counts listed in Table 4 includes only methods that were mutated. Methods that pertain to the enforcement of temporal constraints were ignored. Methods that pertain to the initialization of policies are not mutated as these do not contribute to policy enforcement related activities of X-GTRBAC. Also, methods that pertain to permission-role assignment were not mutated. We did not consider permission role assignments in the case study. In addition, methods that support functionality to be implemented in the future, were ignored. Methods dealing with hierarchy were not mutated because it does not fall into either the A or the I hierarchy.

Eight versions of the policy enforcement module are created by injection of sequence-based malicious faults. Table 5 lists the eight malicious faults $UA \in M$, where M is the set of malicious faults. UA1_1 through UA1_4 correspond to faults whereby the ACUT may deny a user-role activation request that is allowed by the RBAC specification. UA2_1 through UA2_4 correspond to faults whereby the ACUT may allow a user-role activation request that is not allowed by the RBAC specification.

9.2.2 Initialize $Pset$

This is Step 1 in the functional testing technique given in Section 8.1. Policy P_1, shown in Table 6, is initially derived under the pretext that the ACUT is a part of a medical center application. Four roles denoted r_1 through r_4 are considered. The SSoD and DSoD constraints prevent, respectively, roles r_1 and r_2 and roles r_2 and r_3 to be simultaneously assigned to, or activated by, more than one user. The dynamic cardinality constraints on roles (D_r) and users (D_u) are also specified.

Each mutant $f \in F$ created as in Section 9.2.1 was analyzed manually and classified as a UR1, UR2, UA1, or UA2 fault or an equivalent mutant with respect to P_1. The set of equivalent mutants with respect to P_1 is referred as Eqv_{P_1} where $|\text{Eqv}_{P_1}| = 28$. Each $f \in \text{Eqv}_{P_1}$ was manually analyzed to reveal the conditions to distinguish it [16]. Given the complexity of an ACUT, this could turn out to be a rather daunting task as some of these mutants could be semantically equivalent to original program and thus are equivalent with respect to the complete set of RBAC policies $\mathcal{R}$. In the case study two of the 28 equivalent mutants $f \in \text{Eqv}_{P_1}$ were determined to be semantically equivalent to original program and were thus removed from the set of mutants F.

The analysis of remaining 26 mutants helped us in designing P_2, also shown in Table 6, with the precise

Table 5: List of malicious faults injected into X-GTRBAC.

Fault	**Required effect**	**Changes to the code***
UA1_1	P allows activation by virtue of user-role assignment but ACUT does not	Method `activateUserRole` in `GTRBACModule` modified to restrict U_5 activation of an authorized role when U_2 has already activated R_3
UA1_2	P allows activation by virtue of no restriction from $DSoD$ but ACUT does not	Method `checkDSoDValid` modified to prevent activation of (U_3,R_3) pair if $\{R_2, R_3\} \in DSoD$, even when U_3 has not activated R_2.
UA1_3	P allows activation by virtue of no restriction from dynamic user cardinality but ACUT does not	The change is in the `activateUserRole` method in `GTRBACModule`. The fault would reduce the dynamic cardinality of U_2 by one, only under the case if U_2 tries to activate a role after activating R_3 first
UA1_4	P allows activation by virtue of no restriction from dynamic role cardinality but ACUT does not	The change is in the `activateUserRole` method in `GTRBACModule`. The fault would make the check for the validity of role cardinality of the given role to false (even if it is originally true), only when the user activating the given role has already activated R_4
UA2_1	P restricts the given activation by virtue of violation of user-role assignment whereas the ACUT allows	The change is in the `activateUserRole` method in `GTRBACModule`. The fault would allow U_4 to activate role R_2, if not permitted by user-role assignment, for only the cases where U_4 has already activated R_4
UA2_2	P restricts the given activation by virtue of violation of DSoD but the ACUT allows	The change is in the `checkDSoDValid` method of class `Role`. This allows U_3 activation of R_2 even when U_3 has already activated R_3 and $\{R_2, R_3\} \in DSoD$, i.e. although DSoD constraint is violated but the activation request is granted
UA2_3	P restricts the given activation by virtue of violation of dynamic user cardinality but the ACUT allows	The change is in the `activateUserRole` method in `GTRBACModule`. While activating the given role the fault increases the dynamic cardinality of U_4 by one, only when U_4 has already activated R_1 and R_4
UA2_4	P restricts the given activation by virtue of violation of dynamic role cardinality but the ACUT allows	The change is in the `activateUserRole` method in `GTRBACModule`. The fault would make the check for the validity of role cardinality of the given role to true, even when it is originally false, only under the case when U_2 attempts to activate R_2 and U_5 has already activated it, thus violating the role cardinality constraints of R_2

* U_k and R_m, respectively, denote user k and role m

Table 6: Policies P_1 and P_2.

Policy	**Role**	D_r	$SSoD$	S_s	$DSoD$	D_s	**UR assignment**
P_1	Physician (r_1)	3	$\{(r_1, r_2)\}$	1	$\{(r_2, r_3)\}$	1	u_1, u_2, u_4
	Resident (r_2)	1					u_1, u_2, u_5
	Registered nurse (r_3)	3					u_1, u_2, u_4
	Nurse practitioner (r_4)	2					u_4
P_2	Physician (r_1)	1	$\{(r_1, r_2, r_3)\}$	2	$\{(r_1, r_2)\}$	1	u_1
	Resident (r_2)	1	$\{(r_4, r_5)\}$	1	$\{(r_2, r_3, r_4)\}$	2	u_1, u_2
	Surgeon (r_3)	1					u_1, u_2
	Registered nurse (r_4)	1					u_1, u_2
	Nurse practitioner (r_5)	1					u_1, u_2
	Nurse on duty (r_6)	1					u_1, u_2

P_1		P_2	
User	D_u	**User**	D_u
Alice (u_1)	2	Alice (u_1)	2
Bob (u_2)	2	Bob (u_2)	2
John (u_3)	2		
Mary (u_4)	2		
Elie (u_5)	1		

aim of associating these mutants to RBAC faults with respect to P_2. It is to be noted that some of the mutants $f \in F$ would still be equivalent with respect to P_2, however the initial $Pset$ would be considered adequate as for all $f \in F$ the condition $f \in \text{Eqv}_{P_1} \Rightarrow f \notin \text{Eqv}_{P_2}$ could be satisfied. All the faults $f \in F$ can now be classified as a UR1, UR2, UA1, or UA2 fault with respect to either P_1, P_2 or both. The distribution of these faults in various classes in X-GTRBAC is shown in Table 4.

The policy enforcement subsystem makes user-role assignments at the time of policy initialization. Thus only user-role activations and deactivations are performed dynamically. While the code for user-role assignment was mutated, the fault detection effectiveness of procedures used for functional testing would only vary with respect to the simple faults generated by mutating the activation/deactivation code. This included a total of 163 mutants listed under the columns labeled UA1 and UA2 in Table 4. Though the mutants under the columns labeled UR1 and UR2 are used for constructing the $Pset$, they do not effect the fault detection of procedures. Hence UR1 and UR2 mutants are not discussed further in this paper.

9.2.3 Generate $Tset$

This is Step 2 of the functional testing technique given in Section 8.1. Note that each $T \in Tset$ could be generated by using any of the three procedures A, B or C. As the purpose of this case study is to perform comparative analysis of the three procedures, three test sets $Tset_A, Tset_B, Tset_C$ were generated corresponding to the application of procedures A, B and C respectively.

In Procedure A one FSM is generated automatically for each policy in $Pset$. We refer to these as *complete FSM's*, FSM(P_1) and FSM(P_2). As there are no dynamic user-role assignments in X-GTRBAC, FSM(P_1) and FSM(P_2) contain state transitions corresponding only to user-role activations and deactivations. In Procedure B, this characteristic of X-GTRBAC leads to complete FSM's for H1 and H2. There is no FSM corresponding to H3. For H4 there are five FSM's for P_1 and two for P_2, one corresponding to each user. For H5 there are four FSM's for P_1 and six for P_2, one corresponding to each role. As the number of

users was small, we did not group them further, hence H6 was not applied.

As the ACUT state was observable, the transition cover set obtained from testing tree Tr as explained in Section 2.2 served as the test sets $Tset_A$ and $Tset_B$ for procedures A and B. Each test is of the form (r, q), where $r = r_1, r_2, \ldots, r_{k-1}, r_k$, is a sequence of $k > 0$ requests that belong to the input alphabet I and $q = q_1 q_2, \ldots,$
q_{k-1}, q_k is the expected state transition sequence. For Procedure B tests generated by applying H3, H4, and H5 were combined in four different ways: $T(H3) \cup T(H4)$, $T(H3) \cup T(H5)$, $T(H4) \cup T(H5)$, and $T(H3) \cup T(H4) \cup T(H5)$, where $T(x)$ denotes the test set generated by applying heuristic x.

In Procedure C, four pools of 1000 fixed-length tests were generated randomly corresponding to both FSM(P_1) and FSM(P_2). We refer to these pools as RT4, RT6, RT10, and RT100 that contain, respectively, tests of length 4, 6, 10, and 100. These specific lengths were selected as they are comparable with the length of the longest paths in the test tree's for P_1 and P_2 which are 8 and 7 respectively. We considered test sequences of lengths smaller as well as significantly larger than the longest length. Note that tests generated by applying Procedure A vary in length from 1 to 8 for P_1 and 1 to 7 for P_2. However, the distribution by length is not uniform. The advantage of using various pools of fixed length is that it permits investigation of the impact of length of a test suite on its fault detection effectiveness.

A test $t \in$ RTk of length $k \in \{4, 6, 10, 100\}$ is constructed by applying randomly generated requests $r = r_1, r_2, \ldots, r_{k-1}, r_k$ in succession to FSM(P), $P \in \{P_1, P_2\}$ and determining the corresponding state sequence $q = q_1 q_2, \ldots, q_{k-1}, q_k$. Request $r_i, 1 \leq r \leq k$, is generated by selecting randomly user $u \in U$, role $r \in R$ and an input $i \in \{AC, DC\}$ from $P \in \{P_1, P_2\}$.

Five test suites containing 100 tests each were created through random selection of tests from each pool. This led to a total of 20 test suites–each containing 100 tests. We label these test suites as RT$_{ij}$ where $i \in \{4, 6, 10, 100\}$ is the length of each test in the suite, and $1 \leq j \leq 5$, is its identifier. Table 7 shows the number of tests in each test suite generated by applying each of the three test generation procedures to policies P_1 and P_2. The maximum length of tests generated using each of the three heuristics and the complete FSM is listed below. Note that the maximum length of tests generated from H1 and H2 is the same as that of tests generated using the complete FSM.

Policy	**Complete FSM**	**H3**	**H4**	**H5**
P_1	8	40	3	4
P_2	7	24	4	3

9.3 Execute and evaluate tests

This corresponds to Steps 3 and 4 in the functional testing technique. Two test adequacy criteria were used. One criteria is based on mutation. It required that an adequate T_i distinguish all non-equivalent mutants, i.e. T_1 and T_2 should be able to distinguish all $f \in \{F - \text{Eqv}_{P_1}\}$ and $f \in \{F - \text{Eqv}_{P_2}\}$ respectively. The second criteria is based on malicious faults. It required that an adequate T_i detect all malicious faults, i.e. T_1 and T_2 should be able to distinguish all $m \in \{M - \text{Eqv}^m_{P_1}\}$ and $m \in \{M - \text{Eqv}^m_{P_2}\}$ respectively. The set $\text{Eqv}^m_P \in \{P_1, P_2\}$ denote the set of malicious faults equivalent with respect to P.

All non-equivalent mutants and malicious versions of X-GTRBAC were executed against the generated $T \in Tset$. Mutant execution was done automatically by the muJava tool. The tool also reports counts of distinguished and live mutants. Malicious versions were executed using a test harness. The harness integrated three tasks: test generation as describe earlier, test execution by input to a malicious version, and

Table 7: Size of test suites generated using Procedures A, B, and C and policies P_1 and P_2. T_1 is generated from P_1 and T_2 from P_2. $|T|$ denotes the number of elements in set T.

Procedure	**Heuristic**	$\|T_1\|$	$\|T_2\|$	**Comments**
A	None	1,548,847	2,150,05	These tests are generated from complete FSM.
B	H1	1,548,847	2,150,05	FSMs generated using H1 and H2 are identical to complete FSM as X-GTRBAC uses static user-role assignment.
	H2	1,548,847	2,150,05	
	H3	1	1	Only a single sequence of activation and deactivation requests is used as a test. This sequence was generated manually.
	H4	159	849	
	H5	337	63	
	H3+H4	160	850	
	H3+H5	338	64	
	H4+H5	496	912	
	H3+H4+H5	497	913	
C	Random	2000	2000	There are 20 test suites each containing 100 tests. Test suites for a given length i are not necessarily disjoint as these are selected randomly from pool RTi.

response analysis. The observed response was compared against the expected response, the latter being a part of a test case itself.

Under columns labeled P_1 and P_2, Table 8 lists the percentage of simple and malicious faults $f \in \{F - \text{Eqv}_{P_1}\}$, $m \in \{M - \text{Eqv}_{P_1}^m\}$ and $f \in \{F - \text{Eqv}_{P_2}\}$, $m \in \{M - \text{Eqv}_{P_2}^m\}$ detected by test suites T_1 and T_2 respectively for all the procedures. Note that as each RT_i signifies a set of test suites $\{RT_{i1}, RT_{i2} \ldots, RT_{i5}\}$, therefore the fault detection effectiveness of RT_i in Table 8 represents the average effectiveness of test suites $\{RT_{i1}, RT_{i2} \ldots, RT_{i5}\}$.

9.3.1 Enhance $Pset$

This corresponds to Step 5 in the functional testing methodology. Table 9 illustrates the fault detection effectiveness results for the complete test sets $Tset_x = \{T_1, T_2\}, x \in \{A, B, C\}$ corresponding to the usage of three procedures A, B and C. At this point we observe that only the versions of $Tset$ generated using Procedure A i.e. $Tset_A$, and RT_{100} are adequate with respect to both the adequacy criteria. Note that the versions of $Tset_B$ generated by applying H3, H4, and H5 are not adequate with respect to any of the two adequacy criteria. However combining the tests generated using the individual heuristics provides complete fault coverage with respect to simple faults. As we considered our stopping criteria to be based on complete coverage of simple faults, hence functional testing corresponding to all the procedures can be terminated.

If a tester were to use Procedure B, combining the test suites obtained by applying H3, H4 and H5 is the best option. For Procedure C, if RT_{100}, RT_{10} or RT_6 is not used then additional iteration requiring a new policy and starting at step 2 is needed for RT_4. In the case study we terminated functional testing for all three procedures as they were found adequate with respect to our stopping criteria of complete coverage of

Table 8: Fault detection effectiveness (in %) of test suites generated from policies P_1 and P_2.

Procedure	**Heuristic**	**UA1**		**UA2**		**Malicious**	
		P_1	P_2	P_1	P_2	P_1	P_2
A	None	100	100	100	100	100	100
B	H3	98.50	98.60	94.11	97.80	0	0
	H4	96.92	95.65	74.11	84.61	75.00	100
	H5	83.00	88.40	71.76	70.32	25.00	0
	H3+H4	98.46	100	97.64	97.80	75.00	100
	H3+H5	98.46	100	97.64	100	25.00	0
	H4+H5	100	100	98.87	98.90	87.50	100
	H3+H4+H5	100	100	100	100	87.50	100
C	RT_4	91.07	93.04	75.58	87.47	42.50	60.00
	RT_6	100	99.72	97.20	96.48	60.00	100
	RT_{10}	100	100	98.60	99.20	82.50	100
	RT_{100}	100	100	100	100	100	100

Table 9: Fault detection effectiveness (in %) of combined test suites generated using P_1 and P_2.

Procedure	**Heuristic**	**UA1**	**UA2**	**Malicious**
A	None	100	100	100
B	H3	99.00	98.00	0
	H4	96.00	82.00	75.00
	H5	88.00	71.00	25.00
	H3+H4	100	98.00	75.00
	H3+H5	100	100	25.00
	H4+H5	100	99.00	87.50
	H3+H4+H5	100	100	87.50
C	RT_4	94.50	86.75	47.50
	RT_6	100	100	70.00
	RT_{10}	100	100	82.50
	RT_{100}	100	100	100

simple faults.

9.4 Case study: Analysis of results

Number of tests generated: From Table 7 we observe a significant variation in the number of tests generated from the three procedures. As expected, Procedure A generates the largest number of tests–about four thousand orders of magnitude more than those generated using H4 and H5. The maximum length of tests generated using H4 and H5 is also about one-half that of tests generated using the complete FSM. Note that it is by design that only one test is generated when using H3 though it has the maximum of lengths of all tests generated. As discussed later, length of a test impacts its cost and hence the cost benefit ratio of a test generation procedure.

Fault detection effectiveness: We observe from Table 9 that as expected complete FSM based test generation (Procedure A) has 100% fault detection effectiveness for both simple and sequence-based malicious faults. Neither of the individual test suites generated through Procedure B using each of H3, H4, and H5 is adequate with respect to any of the two adequacy criteria. Certainly one would expect this result given the "isolationist" nature of each heuristic and the significantly smaller number of tests generated by these heuristics as compared to the number of tests generated from the complete FSM. Despite this inadequacy, we stopped adding new policies as the combined set of tests generated using H3, H4, and H5 is adequate with respect to simple faults.

Given that the FSM's generated using any heuristic contains only "local" information about a policy, we do not expect any single heuristic to generate a fairly good test suite. However, combining their respective test suites enables the exploitation of locality information across the FSM's that led to an adequate test suite. It is interesting to observe that none of the test suites generated using Procedure B is able to detect all malicious faults. This observation leads to the recommendation that black box testing without the use of code coverage assessment, will likely be unable to detect some malicious faults.

The CRTS strategy: We observe from Table 9 that except for RT_4, all randomly generated test suites are able to achieve complete detection, on the average, of simple faults. What Table 9 does not show explicitly is that for RT_4 at least one of the five pools of 100 tests is unable to detect some simple faults. It can be observed that the fault detection effectiveness of random test suites of same length is higher for an $RT_i \in T_2$ in comparison with corresponding $RT_i \in T_1$. This observation support our assertion (Section 7.3)that random test suites of lengths comparable with the longest test sequence generated using Procedure A (8 for P_1 and 7 for P_2) are expected to have good fault detection. Moreover it can also be easily observed that fault detection effectiveness increases with the increase in length of tests in the CRTS test suites.

Notice from Table 9 that the average effectiveness of randomly selected tests, each of length 6, in detecting UA1 and UA2 faults is the same as that of similarly selected tests of sizes 10 and 100. This observation indicates the existence of an optimal length of test suites that is good enough to obtain adequacy with respect to simple faults. In the case study this length is close to 6. However, there is at least one pool of 100 tests generated randomly each of length 4, 6, and 10, that is unable to detect all malicious faults; those of length 100 did detect all malicious faults. We performed additional experiments to find the least i such that each of the five pools of 100 randomly generated test suite RT_i detects all malicious faults injected. This number turned out to be 26.

Cost-benefit analysis: While the cost of testing consists of several components, here we consider the total number of state variable queries performed in the execution of a test suite as its cost. The cost of $Tset$ generated by a test generation procedure directly depends on lengths of tests in each test suite contained

Table 10: Cost benefit data for Procedures A, B, and C. All values have been rounded to the nearest decimal.

Procedure	**Heuristic**	**Simple Faults**			**Malicious**		
		P_1	P_2	Combined	P_1	P_2	Combined
A	None	1454422	99203	1444727	27452227	15872604	29436303
B	H3	5	2	7	N/A	N/A	N/A
	H4	11	129	130	234	17226	3105
	H5	43	2	41	2477	N/A	2604
	H4+H3	14	111	122	367	17514	3286
	H5+H3	39	3	39	2877	N/A	3148
	H4+H5	42	110	147	908	17480	3406
	H4+H5+H3	47	111	153	1023	17768	3561
C	RT_4	322	167	438	13333	24000	16000
	RT_6	402	231	589	12000	36000	16000
	RT_{10}	666	377	982	14286	60000	22857
	RT_{100}	6622	3750	9816	125000	600000	200000

N/A: Cannot be computed as no faults were detected.

Combined: Cost/benefit ratio for $Tset$ corresponding to $Pset = \{P_1, P_2\}$.

in $Tset$. Recall that each test consists of a sequence of k requests. Hence the length of each test is the number of request it contains. Note that here we ignore the cost of generating additional policies in the test enhancement phase, a largely manual and often a difficult task. We only consider the cost associated with the $Tset$ obtained when testing stops.

Table 10 lists the computed cost/benefit ratio (CBR) for all the procedures used in the case study. CBR is computed as the ratio of cost of a test suite to the number of fault detected by that test suite. It is useful to examine the CBR values in the context of the fault detection effectiveness shown in Tables 8 and 9. While the CBR is the least for H3, it is certainly not a recommended option alone due to its low fault detection effectiveness for malicious faults. Clearly, among the heuristics the CBR for the combination of H3, H4, and H5 is significantly less than that for Procedure A while its fault detection effectiveness is close to that of Procedure A. However, RT_{10} also has a significantly lower CBR as compared to that for Procedure A and almost the same effectiveness as that of the H3, H4, and H5 combination.

Given its high fault detection effectiveness and a cost reduction factor of over 100 against Procedure A, RT_{100} appears to be the best option when Procedure A is impractical. In fact the cost could be reduced further without effecting the effectiveness by reducing the length of randomly generated tests. While this conclusion seems the best for the given case study, we recommend that both random test generation and generation based on a combination of H3, H4, and H5 be used. Doing so CBR remains about 100 times less than that of Procedure A while the risk of faults remaining undetected may reduce.

9.5 Case study: Discussion

Test automation: Manual generation and execution of tests can be costly and error prone. The table below shows automation used in the case study with respect to steps in the proposed functional testing technique described in Section 8.1. Note that Step 3 is partially automated as fault removal requires human intervention. Steps 1 and 5 require manual construction of a new policy. While random generation of policies is feasible, we did not take this course. Instead, policies were generated manually.

Steps	Automated ?
1	No
2	Yes
3	Yes, partially
4	Yes
5	No

What test generation procedure to use?: It is obvious that Procedure A based on complete FSM is likely to be impractical except in environments with a small number of users and roles. As a rule of thumb, Procedure A is not recommended when the number of user-role combinations exceeds 20 leading to about 1.4 million tests.

For most organizations the best strategy seems to be a combination of heuristics based and CRTS strategies. H3 alone is not sufficient, though as discussed later, a combination of H3, H4, and H5 would be a good choice. Again, as a rule of thumb, one may use the length of the longest path from the root of the testing tree generated from the complete FSM as an estimate for the length of randomly generated tests. Given the set of users U, roles R, and permissions Pr, the length of the longest path in the testing tree corresponding to the complete FSM is bounded by $2|U||R| + |Pr||R| + 1$.

The variation in fault detection effectiveness with the length of tests in a CRTS test suite, witnessed in the results given in Table 9, highlights the obvious fact that tests of longer lengths are able to exercise more number of paths in FSM(P). Another observation worth noting is that even a single test case used in H3 is able to provide good coverage of simple faults but fails to detect any malicious fault. Reason being that the effect of simple faults in P is observed across much larger number of paths in FSM(P) as compared to the number of paths across which malicious faults effect.

What heuristics to use ? In the case study we found that the fault detection effectiveness of a combined set of tests generated from H3, H4, and H5 is superior to that of tests generated using any single heuristic. This is likely to be the case in most testing environments that use Procedure B primarily due to the "isolationist" nature of each heuristic. For example, heuristic H4 generates one FSM for each user and thus does not model the dynamic role cardinality constraints. Hence it would not be possible to further improve fault detection of H4 by selecting policies that fully exploit the dynamic role cardinality constraints. The results of the case study support the obvious fact that scaling the model by applying H1 through H6 might have a negative impact on the fault detection effectiveness of the tests generated.

State observability: In the case study we assumed state observability. This led to the use of testing tree as a source for test generation. However, instead of using the testing tree, one could directly generate tests from an FSM model using alternative methods such as transition tour [3] and the UIO [40]. As long as the methods cover all transitions and states, the fault detection effectiveness for simple faults will remain the same as that observed in the case study. However, the fault detection effectiveness for sequence-based malicious faults, as injected in the case study, will not be same.

It is simple to observe from the FSM given in Figure 2, that if a transition tour covers the transitions $\delta(0000, AS_{11})$, $\delta(1000, AS_{21})$ and $\delta(1010, AC_{11})$ corresponding to the path $(0000) \rightarrow (1000) \rightarrow (1010) \rightarrow (1110)$ and does not cover the last transition, i.e. $\delta(1010, AC_{11})$ across the path $(0000) \rightarrow (0010) \rightarrow (1010) \rightarrow (1110)$ then a sequence-based malicious fault that leads to a transfer fault in this transition, only across the later path, will not be detected.

The most effective method to use [43] when states are not observable is the W-method [9]. Achieving state observability might require access to source code. In the absence of such access, one needs to rely on

the W-method that uses the state characterization set to determine if an implementation has indeed moved to the expected state. The fault detection effectiveness of the generated tests will now depend on the accuracy of the estimate of the umber of states in the ACUT [9].

How to enhance $Pset$? The functional testing technique (Section 8.1) requires an initial set $Pset$ of policies, which is enhanced in Step 5 if the adequacy criteria is not satisfied. Various approaches for construction of initial $Pset$ have been already discussed in Section 8. Construction of additional policies, required during the test enhancement phase when $Tset$ is found inadequate, would require a careful analysis of the adequacy data. The analysis would reveal the conditions required to satisfy the criteria, and would lead to a test case t. However, one needs to go a step further and construct a policy that would lead to the generation of t or any other test that satisfies the criteria. Given the complexity of the ACUT, this could turn out to be a rather daunting task. Policies with empty user or role set, and their combinations, might also be useful in checking whether the ACUT implements syntactically valid though practically useless policies.

What Adequacy Criteria to Use ? Step 5 in the functional testing procedure requires that test generation stop when an adequate meta test set $Pset$ and set of test suites $Tset$ has been obtained. While the white-box adequacy criteria used in the case study is based on first order mutations and malicious faults, one could use other criteria in practice. For example, one could use one or more of a number of control flow [48] and data flow [10, 39] based coverage criteria.

A number of formal [10, 21, 34] and case studies point to the fault detection effectiveness of various criteria [20, 37, 47]. Such studies should serve as a guide in making a decision on the stopping criteria. Depending on the availability of resources, one might decide to use the less effective or the more effective of the coverage criteria. Notice that not using a quantitative stopping criteria will likely lead to a weaker $Tset$ as would be the case had we performed testing with $Pset = \{P_1\}$ or $Pset = \{P_2\}$.

Note that a second adequacy criteria in the case study was based on malicious faults. This obviously cannot be used in practice as one does not know in advance whether or not any malicious faults are present in the code. Nevertheless, the difficulty of detecting malicious faults, as is evident from the data in Tables 8 and 9, suggests that at least some form of code based adequacy criteria be used. Certainly, code inspections are also recommended.

The above mentioned approach to stop testing is feasible when code for ACUT is available and can be successfully compiled. If not, then one needs to resort to other approaches. One such approach is based on statistical considerations [14]. We do not have sufficient data to to assess the goodness of such an approach while testing an ACUT. In the absence of such data it becomes important that the ACUT be tested form a variety of policies derived as discussed earlier.

9.6 Case study: Lessons Learned

We consider that case study helped us in learning following key lessons, which could be of particular interest to a practitioner in applying proposed approach for test generation on a RBAC system.

1 Though conformance test generation for RBAC systems using FSM based behavior modeling provides good fault coverage, yet could be prohibitively expensive. A best balance between cost and effectiveness could be obtained by using the CRTS strategy. The CRTS strategy might also be effective in other test generation problems where applications are modeled as FSM but complete test suite cannot be generated because of high cost.

2 The case study reaffirms the well recommended advise for usage of white box criteria in test gener-

ation. Although conformance testing is able to establish ACUT correctness with respect to a single policy but, functional testing requires usage of white box coverage criteria as a feedback mechanism for establishing ACUT performance with respect to all policies.

9.7 Case study: Threats to validity

The threats to validity [4, 46] of our case study are briefly summarized below.

Conclusion Validity: It is related with our ability to draw conclusion about the relation between CBR and fault detection effectiveness of the usage of three procedures in functional testing of an ACUT. The case study used only one initial $Pset$, derived using the program mutation adequacy criteria. The experiment described in the case study could also be conducted by varying the initial $Pset$ and the tests adequacy criteria. This might effect the cost of various procedures and hence the CBR. However, we believe that in any case, the relative CBR of various procedures will likely remain as in Table 10 because heuristics reduce the model size, and hence the size of the test set and the size of randomly generated tests is fixed a priori.

Internal Validity: It is related with the concern that factors other than the variation in test suites can effect the fault detection results for the usage of the three procedures in functional testing of an ACUT. The test suites for all the procedures were executed against the same versions of X-GTRBAC which were either injected with simple or malicious faults. Note that simple faults were injected automatically by muJava. The fault detection for simple faults was measured by automatically executing the test suites from the three procedures against the mutants under a common operating environment. Tests were executed manually against the malicious faults under same operating environment.

External Validity: It is concerned with generalization of our results for other implementations of access control systems. Evaluation of the proposed procedures for test generation was conducted using one implementation, namely X-GTRBAC. Therefore, we cannot generalize the fault detection effectiveness results to other implementations. Further, X-GTRBAC is a stand alone policy enforcement application. While it can be used as a front end to an application, it is not embedded in it. Our case study did not make any use of X-GTRBAC features to actually enforce access control in an application such as a database engine. Fault detection effectiveness of all procedures described might be different than reported in the case study in the event the access control mechanism interacts in complex ways with the application.

The case study evaluates a proposed approach to test generation that is specific to access control implementations that employ RBAC policies. While the proposed approach could be adapted to support testing of other forms of access control, such as DAC and MAC [42], we cannot generalize the results of our case study to implementations of such protocols. Furthermore, new approaches to specifying secure information flow make use of typed programming languages such as Jif [35].

Construct Validity: It is related with the validity of the "constructs" we used for measurement. Fault detection effectiveness of a test suite was measured using the number of both simple and malicious faults detected by that suite. A well known issue in using first-order mutations for effectiveness measurement is whether or not the mutants are representative of real faults. Researchers have found that the use of mutation as a tool for effectiveness evaluation achieves trustworthy results [4, 15]. Nevertheless, in addition to mutants, we also used malicious faults in our case study.

Malicious faults were injected manually without any "real" malicious intention. There certainly exists the possibility that a malicious programmer may inject faults that are much more difficult to find than the ones we injected. It is hard to conduct a case study that would inject "really malicious" faults except when a set of such faults, found in real systems, is available. While data on access control vulnerabilities is

available [26], we do not know what fault in code led to these. Further, these vulnerabilities are not specific to RBAC. Hence, we did not have any "real life" data on malicious faults to consider in the case study.

The cost of a procedure was measured as a function of total length of all the tests in its test suite. Although there are various other factors such as generation of initial set of policies and test enhancement that contributes to the total cost of a procedure but there value would have been same for all the three procedures.

10 Related Work

Ferraiolo and Kuhn [18] proposed Role Based Access Control (RBAC). Although some research has been reported in the verification of RBAC and related policies [1, 17, 32, 25], little has been reported in the testing of software implementations of access control policies. Chandramouli and Blackburn use model-based approach for security functional testing of a commercial database system that employs Discretionary Access Control (DAC) [8]. Their work differs substantially from ours in the creation of tests. In contrast to the FSM based approach proposed here, Chandramouli and Blackburn create tests from system specifications, expressed as Software Cost Reduction (SCR) language, by using predicate based testing approach. Their test generation approach does not consider the issue of determining the fault detection effectiveness of generated tests.

The automata theoretic based approaches for test generation [9, 23] use a FSM model that explicitly captures the expected behavior of the implementation . 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. 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.

Various techniques for state space reduction in FSM based testing and verification have been proposed. Friedman et al. [22] consider a projected state machine of the original FSM from which tests are generated using coverage criteria and test constraints. Heuristic H6 in Section 7.2 is similar to the projected state machine concept used in [22]. Norris and Dill [27] present a state space reduction technique based on structural symmetry information in the system description. Though their primary aim is to aid in verification but the approach can also be used for testing. Test case reduction can also be achieved by a combinatorial approach in which the generated tests ensure coverage of n-way combinations of the test parameters [11].

It is important to note that verification techniques achieve state space reduction by using state abstractions and keep minimal necessary information that can permit verification of a property. However while testing a system, it is essential that complete system information is checked for validity which may not be possible by using state abstraction. To avoid test explosion, an alternate approach in testing could be based on random selection of a fixed number of paths from the non-reduced FSM. We examined both these approaches, in relation with test generation from complete FSM, to determine the impact of state abstraction and random path selection on fault detection effectiveness.

Policy can also be specified as a programmer can embed information flow policies in the program using types that are extensions of existing types, e.g. type `int` is extended in Jif [35] by allowing the declaration to include labels that express policy restrictions. In such cases there is no explicit policy P specification;

the policy is embedded in the implementation. Perhaps such specification is available, or could be derived based on policy requirements.

11 Summary and conclusions

A functional test generation technique that uses one of the three conformance testing procedures is proposed and evaluated. The technique raises the task of testing RBAC implementations from an ad hoc to a formal level so that it can be automated. Exhaustive testing, proposed in Procedure A, of any but the simplest of RBAC policies is impractical when the number of user-role combinations is large (say over 100). Procedure B utilizes state abstraction and heuristics to model the expected behavior of an RBAC implementation. The heuristics lead to a much smaller and, in many cases, practically executable set of tests. Though state abstraction reduces the size of model, it results in a localized view of the system which raises the possibility of undetected faults in the ACUT. In Procedure C, we investigated an alternate approach for test suite reduction by selecting random paths of fixed length from the original non-reduced model of the system.

An empirical evaluation was carried out to assess the cost, fault detection effectiveness, and cost-benefit ratio associated with the usage of the three proposed procedures in functional testing of an ACUT. Two types of faults were injected into a prototype ACUT: first order mutants and malicious faults. Procedure A, as expected, was able to provide complete fault coverage for both the simple and malicious faults, but led to high CBR. Procedure B also achieved complete coverage for simple faults but failed to detect one malicious fault. Despite the low CBR of Procedure B, its use is not recommended when ACUT code is not available and white box coverage measures cannot be used. Procedure C detected all the simple and malicious faults while exhibiting CBR slightly above the CBR of Procedure B.

The heuristics only consider a local view of the system and therefore those faults, that are only exhibited across very small number of paths in the complete FSM are difficult to get detected. The CRTS procedure is better able to execute such paths as it randomly generates the tests through path selection from the complete FSM. The case study indicates that Procedure C can be most effective and cost efficient in the detection of both types of faults in an access control system. However, generalization of the observations for broader range of implementations of access control systems would require further empirical studies and evaluation.

The tests generation strategy proposed is with respect to a definition of RBAC. Though not explained in this paper, the proposed strategy can also handle variations such as various control flow dependency constraints and other non-temporal constraints in the finite state model. Also, the effectiveness evaluation described above is with respect to the proposed RBAC fault model, other fault models could also be designed.

References

[1] T. Ahmed and A. R. Tripathi. Static verification of security requirements in role basedCSCW systems. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 196–203, New York, NY, USA, 2003. ACM Press.

[2] G-J. Ahn and R. Sandhu. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur.*, 3(4):207–226, 2000.

[3] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604–1615, 1991.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 402–411, New York, NY, USA, 2005. ACM Press.

[5] F. Belli and R. Crisan. Towards automation of checklist-based code-reviews. In *Seventh International Symposium on Software Reliability Engineering (ISSRE'96)*, pages 24–33, 1996.

[6] R. Bhatti, A. Ghafoor, E. Bertino, and J. B. D. Joshi. X-GTRBAC: an xml-based policy specification framework and architecture for enterprise-wide access control. *ACM Trans. Inf. Syst. Secur.*, 8(2):187–227, 2005.

[7] G. V. Bochmann, A.Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault models in testing. In *Protocol Test Systems*, pages 17–30, 1991.

[8] R. Chandramouli and M. Blackburn. Automated testing of security functions using a combined model & interface driven approach. In *Proc. 37th Hawaii International Conference on System Sciences*, pages 299–308, 2004.

[9] T. S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[10] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, 1989.

[11] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–89, September 1996.

[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, 2003.

[13] S. R. Dalal and C.L. Mallows. Factor-covering designs for testing software. *Technometrics*, 40(3):234–243, 1998.

[14] S. R. Dalal and A. A. McIntosh. When to stop testing for large software systems with changing code. *IEEE Transactions on Software Engineering*, 20(4):318 – 323, April 1994.

[15] M. Daran and P. Thèvenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 158–171, New York, NY, USA, 1996. ACM Press.

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

[17] M. Drouineaud, M. Bortin, P. Torrini, and K. Sohr. A first step towards formal verification of security policy properties for RBAC. In *Proc. Of Fourth International Conference on Quality Software, (QSIC04)*, pages 60–67, 2004.

[18] D. Ferraiolo and R. Kuhn. Role-based access control. In *Proceedings of the NIST-NSA National (USA) Computer Security Conference*, pages 554–563, 1992.

[19] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.

[20] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[21] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault detection ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, 1993.

[22] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 134–143, New York, NY, USA, 2002. ACM Press.

[23] 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):591–603, June 1991.

[24] A. Gill. *Finite-State Models for Logical Machines*. John Wiley and Sons, Inc., New York, 1968.

[25] F. Hansen and V. Oleshchuk. Lecture notes in computer science. chapter: Conformance checking of rbac policy and its implementation. In R. H. Deng, F.Bao, H-H. Pang, and J. Zhou, editors, *Proceedings of Information Security Practice and Experience: First International Conference, ISPEC 2005, Singapore*, volume Volume 3439 / 2005. Springer Berlin/Heidelberg, 2005.

[26] http://www.cve.mitre.org/. Common vulnerabilities and exposures.

[27] C. N. Ip and David L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9(1-2):41–75, 1996.

[28] J. B. D. Joshi, B. Shafiq, A. Ghafoor, and E. Bertino. Dependencies and separation of duty constraints in GTRBAC. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 51–64, New York, NY, USA, 2003. ACM Press.

[29] M. Koch and F. Parisi-Presicce. Visual specifications of policies and their verification. In *Proc. FASE 2003 (M.Pezze, ed.), Lect.Notes Comp.Sci. 2621*, pages 278–293. Springer-Verlag, 2003.

[30] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, August 2005.

[31] G. Luo, G. V. Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Trans. Software Eng.*, 20(2):149–162, 1994.

[32] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, 1999.

[33] Y-S. Ma, J. Offutt, and Y-R. Kwon. MuJava: an automated class mutation system. *Software Testing, Verification, and Reliability*, 15(2):97–133, 2005.

[34] A. P. Mathur and W. E. Wong. A theoretical comparison between mutation and data flow based test adequacy criteria. In *CSC '94: Proceedings of the 22nd annual ACM computer science conference on Scaling up : meeting the challenge of complexity in real-world computing applications*, pages 38–45, New York, NY, USA, 1994. ACM Press.

[35] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157196, 2006.

[36] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, New Jersey, 2004.

[37] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, 26(2):165–176, 1996.

[38] A. Petrenko, G. v. Bochmann, and M. Yao. On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems*, 29(1):81–106, 1996.

[39] S. Rapps and E. J. Weyuker. Data flow analysis techniques for test data selection. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 272–278, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[40] K. K. Sabnani and A. T. Dahbura. A Protocol Test Generation Procedure. *Computer Networks and ISDN Systems*, 15:285–297, 1988.

[41] R. Sandhu. Role activation hierarchies. In *RBAC '98: Proceedings of the third ACM workshop on Role-based access control*, pages 33–40, New York, NY, USA, 1998. ACM Press.

[42] R. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications*, 32(9):40–48, 1994.

[43] D. P. Sidhu and T. K. Leung. Formal methods for protocol testing: a detailed study. *IEEE Transactions on Networking*, 1(1):116–129, February 1993.

[44] H. H. Thompson. Why security testing is hard. *IEEE Security and Privacy*, 1(4):83–86, 2003.

[45] E. J. Weyuker. In defense of coverage criteria. In *11th International Conference on Software Engineering*, pages 361–361, May 1989.

[46] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Boston, 2000.

[47] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Proceedings of 5th International Symposium on Software Reliability Engineering*, pages 230–238, November 1994.

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