Scalable and Effective Test Generation for Role-Based Access Control Systems

Ammar Masood, Rafae Bhatti, *Member*, *IEEE*, Arif Ghafoor, *Fellow*, *IEEE*, and Aditya Mathur, *Member*, *IEEE Computer Society*

Abstract—Conformance testing procedures for generating tests from the finite state model representation of Role-Based Access Control (RBAC) policies are proposed and evaluated. A test suite generated using one of these 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. Empirical studies revealed that the second approach to test suite generation, combined with one or more heuristics, 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.

Index Terms—Role-Based Access Control (RBAC), finite state models, fault model, first-order mutants, malicious faults.

1 Introduction

Access control 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 be correct in that it faithfully implements the intended policy. Hereafter, an implementation of access control policies is referred to as ACUT (for Access Control Under Test).

The access control policy used in this work is based on the NIST standard Role-Based Access Control (RBAC) model [14], [18], [28]. By an access control policy, we mean a policy that conforms to the standard RBAC model, and therefore has the following features:

1. Definition of Roles,
2. Definition of Permissions,
3. Definition of Users,
4. Mapping of Users to Roles,
5. Mapping of Permission to Roles,
6. Role Hierarchies defined on roles,
7. Constraints defined on user-to-role and permission-to-role mappings.

It should also be clear that our focus in this paper is not the RBAC model itself, but rather one aspect of its administration, i.e., testing of an RBAC implementation.

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 this question the use of a "complete Finite State Machine (FSM)" based conformance testing strategy was investigated. The strategy was to construct an FSM model of the RBAC policy and generate tests from the model using the transition cover set [8]. 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 5.1. The fault coverage of complete FSM strategy is further assessed by extending the RBAC fault model and thus considering the nonmutation faults (referred to 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. Hence, we suggest using white-box coverage criterion such as one based on data flow or 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.

While complete FSM-based conformance testing 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?* We answer this question using two strategies, namely, heuristics-based strategy and constrained random testing. These

- *A. Masood is with the Department of Avionics Engineering, Institute of Avionics and Aeronautics, Air University, E-9 Islamabad, Pakistan. E-mail: ammar.masood@mail.au.edu.pk.*
- *R. Bhatti is with Oracle, 400 Oracle Parkway, Redwood Shores, CA 94065. E-mail: rafae@ieee.org.*
- *A. Ghafoor is with the School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Ave., West Lafayette, IN 47907. E-mail: ghafoor@ecn.purdue.edu.*
- *A. Mathur is with the Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN 47906. E-mail: apm@cs.purdue.edu.*

Manuscript received 19 Mar. 2008; revised 18 Feb. 2009; accepted 20 Mar. 2009; published online 15 May. 2009.
Recommended for acceptance by P. Frankl.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-03-0109.
Digital Object Identifier no. 10.1109/TSE.2009.35.

two strategies are described in Sections 6.2 and 6.3, respectively. We note that abstractions used in the proposed heuristics are also used in formal verification [1], [17].

Contributions.

1. A fault model for RBAC implementations based on selective mutations and on (possibly) malicious code,
2. conformance testing strategy based on complete FSM, heuristic's and Constrained Random Test Selection (CRTS) procedures,
3. a technique for functional testing of ACUT,
4. evaluation of the usage of the three conformance testing procedures in the proposed functional testing technique through a case study.

Organization. RBAC and construction of a finite state model from a given RBAC policy is reviewed in Section 2. After describing the testing context of the proposed test generation methods in Section 3, Section 4 reviews the conformance relation used in the testing procedures. A fault model used for assessing the effectiveness of the tests generated is described in Section 5. We use a mutation-based approach to define a fault model for our RBAC-based testing technique. The proposed conformance test generation procedures are described in Section 6. Section 7 describes the proposed functional testing technique. Section 8 presents salient aspects of an empirical study conducted to assess the cost and fault detection effectiveness of the proposed conformance testing procedures in functional testing of a system. The detailed study is available in [25]. Related work is reviewed in Section 9. Section 10 summarizes the proposed approach.

2 Background

2.1 Role-Based Access Control

The RBAC model [14] is an ANSI standard for access control. 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 nonnegative 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, and
- $D_s : DSoD \rightarrow Z^+$ specifies the cardinality of the $DSoD$ sets.

Each element of policy P is parameterized when there is a need to distinguish it from that of another policy P'. For example, $UR(P)$ and $UR(P')$ are the UR assignments corresponding, respectively, to policies P and P'. $(u, r) \in UR$ implies that an assignment of u to r is allowable, which is distinct from the assignment defined in standard RBAC. The reason we consider (u, r) as only an allowable assignment is that this assignment in our model is conditional upon satisfaction of associated constraints. In particular, u is in fact assigned to r only when 1) an input request $AS(u, r)$ is received and 2) the static user and role cardinality and SoD constraints are satisfied at the time the assignment request is received.

In addition, we also distinguish assignment from activation, which is another notion not found in standard RBAC. In our model, the existence of an assignment (u, r) is not necessarily sufficient to let u activate r. Instead, for user u to be authorized to activate role r, 1) input request $AC(u, r)$ must be received, 2) u must be assigned to r or permitted via $\leq_A$, and 3) dynamic user and role cardinality and SoD constraints must be satisfied. This is a key distinction between our model and the standard RBAC, and is vital to introduce the constraint specification mechanism that we support in our policy.

Building on these two distinctions, we similarly extend the notion of separation of duty constraints. 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. 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 finite state model of a given RBAC policy, described in Section 2.2, can be extended to model the user sessions. Administrative constraints can also be integrated in the model. Three types of control flow dependency constraints have been considered in [22] 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. A sample policy follows.

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

$U = \{\text{John}, \text{Mary}\} = \{u_1, u_2\}, R = \{\text{Customer}\} = \{r_1\}, Pr = \{\text{Deposit}, \text{Withdrawal}\} = \{p_1, p_2\}, UR = \{(u_1, r_1), (u_2, r_1)\}, PR = \{(p_1, r_1), (p_2, r_1)\}, S_u(u_1) = S_u(u_2) = D_u(u_1) = D_u(u_2) = 1, S_r(r_1) = 2, D_r(r_1) = 1, \leq_A = \leq_I = \{\}$.

As shown above, each permission p_1 and p_2 in Pr is associated with functions and resources related to the application under consideration.

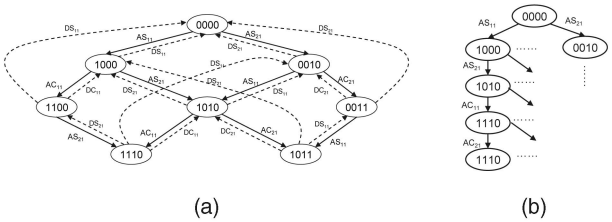


Fig. 1. (a) A complete finite state behavioral model derived from the RBAC policy in Example 1 and (b) its partial testing tree. Expected response is not shown. For each transition between two states, the response is *granted*. Self-loops corresponding to *denied* response are not shown to keep the figure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

2.2 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 corresponding to a specific policy P as an FSM $M = \text{FSM}(P)$. 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. In M AS, DS, AC, and DC are, respectively, abbreviations used 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 and *granted* and *denied* denote two possible responses of an ACUT to any of the four request types. The permission-role assignments are ignored to simplify the presentation.

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

Fig. 1a shows an FSM model M with eight states that represents the behavior of the ACUT required to implement policy P in Example 1. 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 6.2, we propose heuristics to reduce the size of the model and hence that of the test set.

3 Testing Context

Fig. 2 shows the context of the applicability of the proposed test generation approach. First, the ACUT is initialized with a policy P using the Policy processor and an internal representation is constructed. Next, a request received by the ACUT is authenticated against the policy and, if *granted*, passed to the Application.

The Test Harness in Fig. 2 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_i to role r_j."

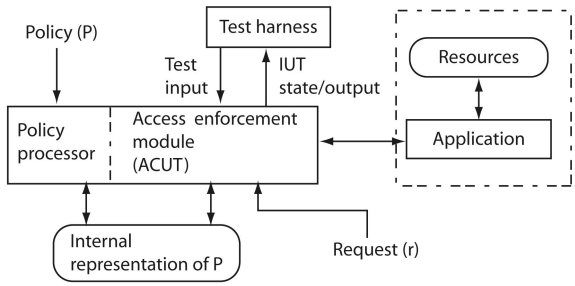


Fig. 2. Interaction between an application, access control enforcement module (ACUT), and the protected resources. Test harness contains test cases generated using a finite state model. Test cases are to test the policy enforcement mechanism, not the application.

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.

4 Conformance Relation

Let $\mathcal{R}$ denote the set of all RBAC policies, O an organization that uses role-based access control to protect its resources, and ACUT′ an implementation used by O 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, O 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 an implementation of a policy P is free of faults that correspond to policies different from P. The proposed fault model is derived with this goal in view. Fig. 3 illustrates the proposed conformance relationship among 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,

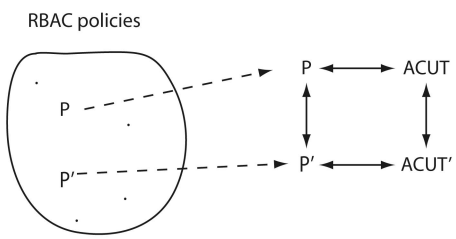


Fig. 3. Policy and implementation conformance. An arrow, in either direction, is to be read as "conforms to."

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

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 combined treatment of user-role assignments, user-role activations, and permission-role assignments through $Rq(up, r)$ is though nonstandard, yet has been done to simplify the presentation of conditions for behavioral conformance.

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}. Note that, for ill-formed requests, $S_{\text{ACUT}}[Rq(up, r)] = S_{\text{ACUT}}$.

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 $Rq(up, r) \in I$, if $S'_{\text{ACUT}} = S_{\text{ACUT}}[Rq(up, r)]$, then $S'_{\text{ACUT}'} = S'_{\text{ACUT}} = S_{\text{ACUT}'}[Rq(up, r)]$.
2. For all ill-formed request $Rq(up, r)$, $S_{\text{ACUT}'}[Rq(up, r)] = S_{\text{ACUT}'}$.

For simplicity, the above conditions do not consider conformance with respect to outputs.

5 RBAC Fault Model

Given a policy $P \in \mathcal{R}$, the set $\mathcal{R}$ can be partitioned into two subsets: conforming ($\mathcal{R}^P_{conf}$) and faulty ($\mathcal{R}^P_{fault}$). Conformance testing of ACUT′ implies verifying that P' does not belong to the set of faulty policies, i.e., $P' \notin \mathcal{R}^P_{fault}$.

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 mutation operators to 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, S_s, and D_s. Three types of set mutation operators are considered: modification of an element, addition of an element, and removal of an element. The application of the 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 $z - 1$.

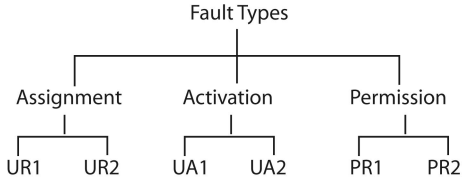


Fig. 4. A fault model for evaluating the effectiveness of tests of RBAC implementations.

For an element $(u, r) \in UR$, the effect of the modification operator could be to 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 in policy P' with possible faults in 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 Fig. 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 fault restricts a permission being assigned to an authorized role or cause an unauthorized deassignment. PR2 fault assigns a permission to an unauthorized role. UA1 and UA2 faults are similar to UR1 and UR2 and impact role activation.

5.1 Relation between FSM and RBAC Fault Model

Chow's fault model for finite state machines [8] consists of four faults: operation, transfer, extra state, and missing state. An operation fault occurs when implementation FSM has a transition with an incorrect input or output label. A transfer fault occurs when implementation FSM has a transition that terminates at an incorrect state. A missing (extra) state fault implies presence of missing (extra) state in the implementation FSM. Table 2 shows the correspondence between the RBAC fault model in Fig. 4 and the one proposed by Chow for FSM. Fig. 5 shows an example that relates UA2 and UR1 faults to the corresponding faults in an FSM.

TABLE 2
Correspondence between the RBAC and FSM Fault Model

RBAC fault model	FSM fault model[8]
UR1, UA1, PR1	Transfer fault, Missing state fault, Output fault
UR2, UA2, PR2	Extra state fault, Output fault, Transfer fault

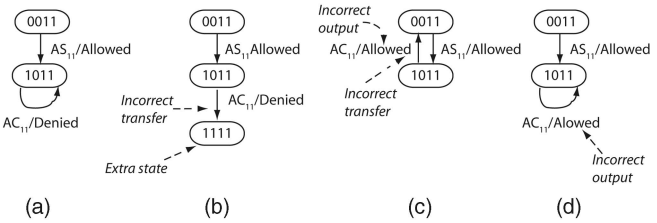


Fig. 5. Mapping of the UA2 and UR1 faults to those in FSM fault model. (a) Correct transitions extracted from Fig. 1. (b) Extra state and transfer fault. (c) Transfer and output faults. (d) Output fault that does not correspond to the proposed fault model because the ACUT remains in a consistent state (1011).

5.2 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 [11], a malicious programmer may inject faults that simulate devious ways of tricking an ACUT into malicious behavior. Next, we consider three types of malicious faults.

Counter-based faults. A counter-based fault exists in an ACUT if it contains reachable code that incorrectly grants, denies, or aborts a request based on counts of events.

I/O-based faults. An I/O-based fault exists in an ACUT if it has reachable code that incorrectly grants, denies, or aborts a malformed request. We ignore faults based on the output alphabet as these can be detected easily by a test harness.

Sequence-based faults. A programmer could 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 has been activated. 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.

Sequence-based faults cannot be related to FSM faults as they are the result of non-FSM behavior of the ACUT. An example of sequence-based fault corresponding to FSM of Fig. 1 is given in Fig. 6a. The input sequences AS_{21}, AC_{11} and AS_{11}, AC_{11} given in the states (1000) and (0010), respectively, of $\mathrm{FSM}(P)$ would always lead to the state (1110), whereas the later sequence would lead to the state (1011) in the $\mathrm{FSM}(P')$. Counter-based faults can be related with extra state FSM fault, as illustrated in Fig. 6b. The input sequence $AC_{11}, DC_{11}, AC_{11}$ given in state (1010) of $\mathrm{FSM}(P)$ would always lead to the state (1110), whereas the same sequence applied in state (1010) of $\mathrm{FSM}(P')$ would lead to state (1011). I/O-based faults cannot be related to FSM faults.

6 Generation of Conformance Test Suite

In this section, we propose four 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 upper bound on cost is measured in terms of the total number of state variable queries performed in the execution of a test suite. The cost effectively depends upon the total number of tests, their lengths, and the number of state variables. The fault

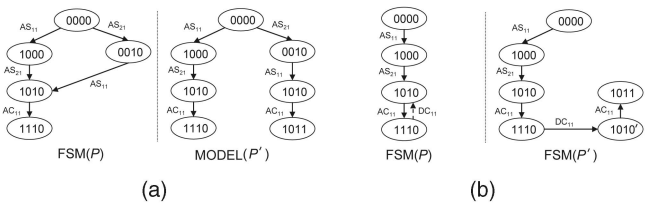


Fig. 6. (a) Example of a sequence-based fault. (b) Mapping of counter-based fault to Chow's fault model.

detection effectiveness of these procedures is evaluated with respect to the faults in the RBAC fault model proposed in Section 5. It is assumed that the policy P' implemented by the ACUT can be modeled as a finite state machine $\mathrm{FSM}(P')$ and ACUT states are observable.

6.1 Procedure A: Complete FSM Based

In this procedure, tests are generated from the complete FSM ($M = \mathrm{FSM}(P)$) by using a transition cover set that includes paths from the root of the testing tree to each leaf [8]. For the $\mathrm{FSM}(P)$ in Fig. 1a, the partial testing tree is shown in Fig. 1b. The upper bound on the number of states in M and of the test execution cost of this procedure is given in Table 4.

State observability. When the states in the ACUT are not observable, the W-method proposed by Chow [8] can be used to construct the conformance test suite. The W-method is based on concatenation of the sequences obtained from the testing tree with the sequences in the state characterization set to determine whether correct state has been reached. Chow has shown that, given an accurate estimate of the number of states in implementation FSM, the W-method detects all faults in the FSM fault model, which, in the case of observable states of $M' = \mathrm{FSM}(P')$, implies that the test suite generated from the transition cover set detects all faults in the FSM fault model. Thus, based on mapping between the simple faults in the proposed RBAC fault model and the FSM fault model, established in Section 5.1, it is claimed that Procedure A guarantees a test set that detects all simple faults in the ACUT that correspond to the proposed RBAC fault model.

For malicious faults, it can be observed that Procedure A is unable to detect I/O-based faults. 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 testing tree contains at least one test case for each path in the FSM. Table 3 lists a sample of tests generated by assuming state observability and the faults each test case is able to detect.

6.2 Procedure B: Heuristics Based

In this procedure six heuristics, labeled H1-H6, are used to reduce the size of the model and of the test set.

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. Figs. 7a and 7b show, respectively, M_{AS} and $M_{AC_{11}}$ for the policy in Example 1. A state in M_{AS} corresponds to assignments whereas one in

TABLE 3
A Sample of Test Inputs Obtained from the FSM in Fig. 1

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 $D_r(r_1) = 1$.

M_{AC} corresponds to activations. For the model in Fig. 1, application of this heuristic leads to an increase in the total number of states from 8 to 12 and a twofold reduction in the number of tests as in Table 4.

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 Fig. 7a. 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, deactivation, 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.

H4: FSM for each user. Construct M_u for each $u \in U$. Apply the test generation procedure separately to each model. Fig. 7b shows 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 Fig. 1. 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.

In addition to the six heuristics in Table 4, 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 4 show, respectively, the number of tests generated when the completeness assumption is relaxed.

Impact on fault detection effectiveness. It is obvious that scaling down the model by applying one or more proposed heuristics might have a negative impact on the fault detection effectiveness of the tests generated. Quantifying this impact for arbitrary implementations is possible only in specific instances through empirical studies. Here, we briefly review the impact due to each heuristic, considering their application to FSM of Example 1.

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 Fig. 7a, 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-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 1,000 and $S_r(r_1) = 750$, a faulty ACUT might allow 1) F1: r_1 to be assigned to more than 750 users or 2) 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, boundary value testing can be used to detect F1 and similar faults related to

TABLE 4
Sizes of Test Sets Obtained by Applying Various Heuristics

Heuristic	Upper bound on		$\lvert Tset\rvert$ for Example 1		
	$\lvert Q\rvert$	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	$T(T+1)(2T)^{T+1}+4T^2$	11	9	5
H3	No FSM	$8T^2$	1	1	1
H4	$\lvert U\rvert 3^{\lvert R\rvert}$	$2T\lvert U\rvert(2\lvert R\rvert+1)(4\lvert R\rvert)^{2\lvert R\rvert+1}$	20	14	8
H5	$\lvert R\rvert 3^{\lvert U\rvert}$	$2T\lvert R\rvert(2\lvert U\rvert+1)(4\lvert U\rvert)^{2\lvert U\rvert+1}$	92	64	40
H6	3^{T_g}	$2T_g(2T_g+1)(4T_g)^{2T_g+1}$	10	7	4

$\lvert X\rvert$ *denotes the number of elements in set* X. $T = \lvert U\rvert \times \lvert R\rvert, T_g = \lvert UG\rvert \times \lvert R\rvert$.

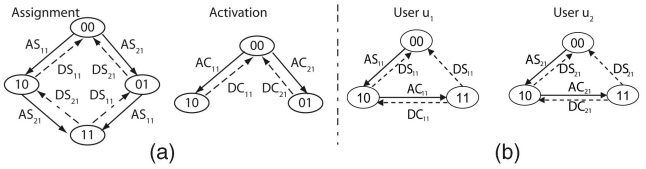


Fig. 7. Models constructed by applying (a) H1 and (b) H4. As in Fig. 1, self-loops and outputs are not shown. Note that for H1 there are four activation FSMs though only $M_{AC_{11}}$ is shown here.

cardinality constraints. The boundary points are derived from the cardinality constraints.

6.3 Procedure C: The CRTS Strategy

The CRTS strategy is to select tests of length $k > 0$ randomly and is aimed at reducing the number of test sequences without requiring reduction in the model size. A test case of length k is constructed by randomly applying generated sequence of requests $r = r_1, r_2, \ldots, r_{k-1}, r_k$ to FSM(P) 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 randomly selecting user $u \in U$, role $r \in R$, and an input $i \in \{AS, DS, AC, DC, AP, DP\}$ from P.

It is suggested that length k be at least equal to the length of the longest path in the test tree corresponding to FSM(P). This allows tests to traverse a complete path from the root of the tree (corresponding to the start state of the FSM) to the leaf. It is easy to see that the length of the longest path in the testing tree, and hence k corresponding to the complete FSM is bounded by $2|U||R| + |Pr||R| + 1$.

6.4 Procedure D: Combining Heuristics and CRTS

Let TS denote the set of all transitions in FSM(P). Application of any of the heuristics mentioned earlier partitions TS. Fig. 8 illustrates this partitioning for some FSM(P) into two subsets denoted as "Hx" and "not Hx (nHx)." Subset Hx, above the horizontal line, contains transitions corresponding to the FSM generated by applying heuristic Hx, and subset nHX below the horizontal line contains the remaining transitions.

Similarly, the CRTS strategy partitions the transition space into two subsets as in Fig. 8: The transitions inside the blob labeled RTy are covered by the tests generated using CRTS and those outside the blob are not covered. Note that the transition coverage could vary even for the same number and length of tests in all the CRTS suites. Assuming state observability, all types of FSM faults can be associated with any given transition. Thus, a test case executing that transition will be able to detect the corresponding fault.

As shown in Fig. 8, Hx is able to detect faults f1 and f3 but misses f2. Fig. 8a illustrates that the combined test suite resulting from the addition of CRTS and Hx suites also includes the transition corresponding to f2 and hence the fault is detected. Figs. 8b and 8c represent the cases where the combined suite is unable to detect all faults as, in Fig. 8b, CRTS-based tests miss both faults f1 and f2 and, in Fig. 8c, miss all of the faults.

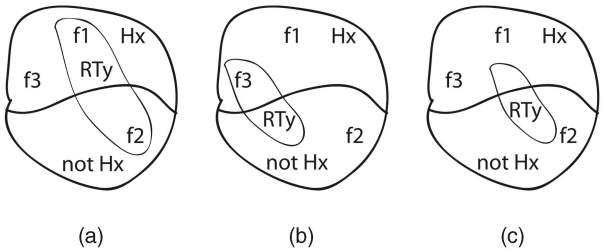


Fig. 8. Faults detected and missed by using strategies based exclusively on heuristics or CRTS. Tests generated from FSM generated using heuristic Hx detect faults f1 and f3 but misses fault f2. Tests generated using CRTS misses (a) fault f3, (b) faults f1 and f2, and (c) all faults.

7 Functional Testing of ACUT

Conformance testing of ACUT establishes its conformance with respect to a specific RBAC policy. Functional testing is required to ensure that ACUT will correctly enforce all policies. Given that the set $\mathcal{R}$ of all RBAC policies is infinite, functional testing implies guaranteeing ACUT conformance with respect to all policies in $\mathcal{R}$. As exhaustive testing is not a viable option, only a 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 criterion such as mutation or code coverage is required to establish correctness of ACUT.

7.1 Proposed Functional Testing Methodology

The functional test suite for an ACUT is a pair $\langle Pset, Tset \rangle$, 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. 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 criterion not satisfied. Go to step 3.

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

7.2 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).

When program mutation is used as an adequacy criterion then mutants can be used in constructing the initial policy set, as is the approach (Section 8.2.2) used in

TABLE 5
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

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 the set of all nonequivalent 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 nonequivalent 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 $P \in Pset$.

8 Empirical Evaluation

An empirical study was conducted to assess the fault detection effectiveness and cost-benefit ratio (CBR) associated with each of the four procedures described in Section 6. The cost is measured as the total number of state variable queries performed in the execution of a test suite. Program mutation [11] and manual injection of malicious faults were used to measure the fault detection effectiveness. The cost benefit is measured as the ratio of the cost of the tests generated using a procedure to the number of faults detected by the generated tests. The study was based on an implementation of a general purpose access control framework named X-GTRBAC [6]. Brief description of X-GTRBAC and details of the study follow.

8.1 The X-GTRBAC Framework

X-GTRBAC is an access control policy specification framework with an associated enforcement mechanism. The access control portion consists of two sets of modules: a policy initializer and a policy enforcer. The initializer loads an RBAC policy coded in XML and the enforcer either allows or denies requests for user-role activations and deactivations based on the policy constraints. Only the policy enforcement subsystem of X-GTRBAC was the target of the case study. This subsystem consists of seven classes listed in Table 5. 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.

8.2 Method and Results

The case study followed the steps for functional testing of ACUT as described in Section 7.1. The steps and results obtained follow.

8.2.1 Configure X-GTRBAC

We used program mutation as adequacy criterion and thus, as described earlier in Section 7.2, $Pset$ was obtained using analysis of the nonequivalent mutants. Section 8.2.2 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 [11], hereafter referred to as simple faults, and sequence-based malicious faults (Section 5.2). The set of program mutants is referred as F.

The policy enforcer module is mutated by applying all Java mutation operators to the classes listed in Table 5. Mutation operators as defined in the muJava system are used [24]. These include the five traditional operators and 23 Java class related operators. The counts listed in Table 5 includes only methods that were mutated. Methods that pertain to the enforcement of temporal constraints and initialization of policies were ignored. Also, methods that pertain to permission-role assignment were not mutated. We did not consider permission-role assignments in the case study. Methods dealing with role hierarchy were also not mutated.

Eight versions of the policy enforcement module are created by injection of sequence-based malicious faults. Table 6 lists the eight malicious faults $UA \in M$, where M is the set of malicious faults. UA1_1-UA1_4 correspond to faults whereby the ACUT may deny a user-role activation request that is allowed by the RBAC specification. UA2_1-UA2_4 correspond to faults whereby the ACUT may allow a user-role activation request that is not allowed by the RBAC specification.

8.2.2 Initialize $Pset$

Policy P_1, shown in Table 7, is derived under the pretext that the ACUT is a part of a medical center application. Four roles, denoted r_1-r_4, are considered. The SSoD and DSoD constraints prevent, respectively, roles r_1 and r_2 and roles r_2 and r_3 being 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.

TABLE 6
List of Malicious Faults Injected into X-GTRBAC

Fault	Required effect on P'	Changes to the code*
UA1_1	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	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	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	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, only when the user activating the given role has already activated R_4
UA2_1	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	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. despite the violation of DSoD constraint the activation request is granted
UA2_3	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	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, 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.

Each mutant $f \in F$ created as in Section 8.2.1 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 [11]. 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 in Table 7, with the precise aim of associating these mutants to RBAC faults with respect to P_2. 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 5.

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 5. 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.

TABLE 7
Policies P_1 and P_2

Policy	Role	D_r	$SSoD$	S_s	$DSoD$	D_s	UR assignment	User	D_u
P_1	Physician (r_1)	3	$\{(r_1, r_2)\}$	1	$\{(r_2, r_3)\}$	1	u_1, u_2, u_4	Alice (u_1)	2
	Resident (r_2)	1					u_1, u_2, u_5	Bob (u_2)	2
	Registered nurse (r_3)	3					u_1, u_2, u_4	John (u_3)	2
	Nurse practitioner (r_4)	2					u_4	Mary (u_4)	2
								Elie (u_5)	1
P_2	Physician (r_1)	1	$\{(r_1, r_2, r_3)\}$	2	$\{(r_1, r_2)\}$	1	u_1	Alice (u_1)	2
	Resident (r_2)	1	$\{(r_4, r_5)\}$	1	$\{(r_2, r_3, r_4)\}$	2	u_1, u_2	Bob (u_2)	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		

TABLE 8
Maximum Length of Tests

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

8.2.3 Generate $Tset$

In Procedure A, one FSM is generated automatically for each policy in $Pset$. We refer to these as *complete FSMs*, $\mathrm{FSM}(P_1)$ and $\mathrm{FSM}(P_2)$. As there are no dynamic user-role assignments in X-GTRBAC, $\mathrm{FSM}(P_1)$ and $\mathrm{FSM}(P_2)$ contain state transitions corresponding only to user-role activations and deactivations. In Procedure B, this characteristic of X-GTRBAC leads to complete FSMs for H1 and H2. There is no FSM corresponding to H3. For H4, there are five FSMs for P_1 and two for P_2, one corresponding to each user. For H5, there are four FSMs 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, as explained in Section 6.1, served as the test sets $Tset_A$ and $Tset_B$ for procedures A and B. For Procedure B, tests generated by applying H3-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 1,000 fixed-length tests were generated randomly corresponding to both $\mathrm{FSM}(P_1)$ and $\mathrm{FSM}(P_2)$. These pools are referred to 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 trees 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.

A test $t \in \mathrm{RT}k$ 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 $\mathrm{FSM}(P), P \in \{P_1, P_2\}$ 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 9 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 given in Table 8. Note that the maximum length of tests generated from H1 and H2 is the same as that of tests generated using the complete FSM.

8.3 Execute and Evaluate Tests

Two test adequacy criteria were used. The mutation-based criterion required that an adequate T_i distinguish all nonequivalent mutants, i.e., T_1 and T_2 should be able to distinguish all $f \in F - \mathrm{Eqv}_{P_1}$ and $f \in F - \mathrm{Eqv}_{P_2}$, respectively. The second criterion, based on malicious faults, required that an adequate T_i detect all malicious faults, i.e., T_1 and T_2 should be able to distinguish all $m \in M - \mathrm{Eqv}^m_{P_1}$ and $m \in M - \mathrm{Eqv}^m_{P_2}$, respectively. $\mathrm{Eqv}^m_P, P \in \{P_1, P_2\}$ denotes the set of malicious faults equivalent with respect to P.

Under columns labeled P_1 and P_2, Table 10 lists the percentage of simple and malicious faults $f \in F - \mathrm{Eqv}_{P_1}$, $m \in M - \mathrm{Eqv}^m_{P_1}$ and $f \in F - \mathrm{Eqv}_{P_2}$, $m \in M - \mathrm{Eqv}^m_{P_2}$ detected by test suites T_1 and T_2, respectively, for all of 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 10 represents the average effectiveness of test suites $\{RT_{i1}, RT_{i2} \ldots, RT_{i5}\}$.

8.3.1 Enhance $Pset$

Table 11 contains the fault detection effectiveness results for the complete test sets $Tset_x = \{T_1, T_2\}, x \in \{A, B, C\}$ corresponding to procedures A, B, and C. 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 test adequacy criteria. Note that the versions of $Tset_B$ generated

TABLE 9
Size of Test Suites Generated Using Procedures A, B, and C and Policies P_1 and P_2

Procedure	Heuristic	$\lvert T_1 \rvert$	$\lvert T_2 \rvert$	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	500	500	There are 5 test suites for a given length i each containing 100 tests. Test suites for a given length i are not necessarily disjoint as these are selected randomly from pool RTi.

T_1 *is generated from* P_1 *and* T_2 *from* P_2.

TABLE 10
Fault Detection Effectiveness (in Percent) 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	98.5	98.6	94.1	97.8	0	0
	H4	96.9	95.6	74.1	84.6	75	100
	H5	83	88.4	71.8	70.3	25	0
	H3+H4	98.5	100	97.6	97.8	75	100
	H3+H5	98.5	100	97.6	100	25	0
	H4+H5	100	100	98.9	98.9	87.5	100
	H3+H4+H5	100	100	100	100	87.5	100
C	RT_4	91.1	93	75.6	87.5	42.5	60
	RT_6	100	99.7	97.2	96.5	60	100
	RT_{10}	100	100	98.6	99.2	82.5	100
	RT_{100}	100	100	100	100	100	100

TABLE 11
Fault Detection Effectiveness (in Percent) of Combined Test Suites Generated Using P_1 and P_2

Procedure	Heuristic	UA1	UA2	Malicious
A	None	100	100	100
B	H3	99	98	0
	H4	96	82	75
	H5	88	71	25
	H3+H4	100	98	75
	H3+H5	100	100	25
	H4+H5	100	99	87.5
	H3+H4+H5	100	100	87.5
C	RT_4	94.5	86.7	47.5
	RT_6	100	100	70
	RT_{10}	100	100	82.5
	RT_{100}	100	100	100

by applying H3-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.

If a tester were to use Procedure B, combining the test suites obtained by applying H3-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 criterion of complete coverage of simple faults.

8.4 Analysis of Results

Number of tests generated. From Table 9, we observe a significant variation in the number of tests generated from the three procedures. Procedure A generates the largest number of tests—about 4,000 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 is the longest of all tests generated.

Fault detection effectiveness. We observe from Table 11 that complete FSM-based test generation (Procedure A) has 100 percent 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-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 in comparison with the one generated from the complete FSM. Despite this inadequacy, we stopped adding new policies as the combined set of tests generated using H3-H5 is adequate with respect to simple faults.

The CRTS strategy. We observe from Table 11 that, except for RT_4, all randomly generated test suites are able to achieve complete detection, on average, of simple faults. What Table 11 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 $RT_i \subset T_2$ in comparison with corresponding $RT_i \subset T_1$. This observation supports our assertion (Section 6.3) 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.

Notice from Table 11 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. We consider the cost of testing to be the total number of state variable queries performed in the execution of a test suite. The cost of $Tset$ directly depends on the lengths of tests in each test suite contained in $Tset$. 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 12 lists the computed CBR for all the procedures used in the case study. CBR of a test suite is defined as the ratio of cost of the test suite and the total number of faults detected by that test suite. It is useful to examine the CBR values in the context of the fault detection effectiveness shown in Tables 10 and 11. 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. The CBR for the combination of H3-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 tests derived from a combination of H3-H5.

Given its high fault detection effectiveness and a cost reduction factor of over 700 against Procedure A, RT_{100} appears to be the best option when Procedure A is impractical. While this conclusion seems the best for the given case study, we recommend that both CRTS and a

TABLE 12
Cost Benefit for Procedures A-C

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 (with duplicates)	RT_4	65	34	88	2353	8000	3368
	RT_6	81	47	118	2500	7200	3429
	RT_{10}	134	76	197	3030	12000	4848
	RT_{100}	1325	750	1964	25000	120000	40000
C (without duplicates)	RT_4	62	33	85	2258	7840	3259
	RT_6	78	46	114	2400	6984	3305
	RT_{10}	129	73	189	2909	11520	4654
	RT_{100}	1272	735	1900	24000	117600	38700

N/A: Cannot be computed as no faults were detected.
Combined: Cost/benefit ratio for $Tset$ corresponding to $Pset = \{P_1, P_2\}$.
All values are rounded to the nearest decimal.

combination of H3-H5 be used. By doing so, CBR remains about 700 times less than that of Procedure A while the risk of faults remaining undetected may reduce. In fact, the cost could be reduced further without affecting the fault detection effectiveness by removing the duplicate tests from the CRTS test suites as illustrated by CBR values given for Procedure C (without duplicates) in Table 12, obtained by considering only distinct tests in the CRTS test suites.

8.5 Impact of Combining Heuristics and CRTS-Based Strategies

We examined the fault detection effectiveness and the cost-benefit ratio for Procedure D. Table 13 contains the results for the combination of RT4, RT6, and RT10 with the heuristics H3-H5. Combining RT4 and RT6 with the heuristics leads to increased fault detection whereby the CBR also increases in most of the cases. For example, RT4+H4 has 91 percent effectiveness for P_1 with a CBR of 68, whereas RT4 alone has an effectiveness of 83 percent with a CBR of 65. In contrast, the combined use of RT10 and heuristics does not increase the fault detection effectiveness though the CBR increases. As RT10 alone has a much higher fault detection effectiveness as compared to that obtained through the application of heuristics, the combined use does not alter the effectiveness whereas the CBR increases due to merger of the two suites. The results in Table 13 indicate that the effect on fault detection effectiveness, as a consequence of the addition of test suites corresponding to heuristics and the CRTS suites, depends on the length of the tests in the CRTS suite. For smaller length suites generated using CRTS, one can expect

TABLE 13
Fault Detection Effectiveness (Effec. in Percent) and Cost-Benefit Ratio for Combined Test Suites Generated by Applying Heuristics and the CRTS Strategy

Technique	P_1				P_2			
	Simple		Malicious		Simple		Malicious	
	Effec.	CBR	Effec.	CBR	Effec.	CBR	Effec.	CBR
H3	98	5	0	N/A	96	2	0	N/A
H4	84	11	75	234	89	129	100	17226
H5	72	43	25	2477	78	2	0	N/A
RT4	83	65	42.5	2353	90	34	60	8000
RT4+H3	98	60	42.5	2588	97	32	60	8488
RT4+H4	91	68	90	1306	98	140	100	22026
RT4+H5	86	100	47.5	3409	93	34	60	8423
RT6	98.4	81	60	2500	98	47	100	7200
RT6+H3	99.6	85	60	2667	99.4	47	100	7488
RT6+H4	99.7	89	95	1764	99.1	154	100	24426
RT6+H5	98.7	114	60	3532	98.5	47	100	7454
RT10	99.2	134	82.5	3030	99.4	76	100	12000
RT10+H3	99.2	139	82.5	3151	99.4	77	100	12288
RT10+H4	99.2	143	92.5	2892	99.4	184	100	29226
RT10+H5	99.2	167	87.5	3565	99.4	77	100	12254

All values are rounded to the nearest decimal.

gains in the fault detection effectiveness. However, this gain reduces as the length of the CRTS generated tests increases.

8.6 Discussion

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. For most organizations, the best strategy seems to be a combination of heuristics and CRTS-based strategies.

What heuristics to use? The fault detection effectiveness of a combined set of tests generated from H3-H5 appears to be 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-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 [26]. 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, may change. The most effective method to use [29] when states are not observable is the W-method [8]. 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 [8].

How to enhance *Pset*? 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 criterion, 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 criterion. 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 criterion to use? Step 5 in the functional testing procedure requires that test generation stops when an adequate meta test set $Pset$ and set of test suites $Tset$ has been obtained. While the white-box adequacy criterion used in the case study is based on first-order mutations and malicious faults, in practice one could use other criteria [30].

8.7 Threats to Validity

Conclusion validity. The case study used only one initial $Pset$ and one software artifact (X-GTRBAC). The experiment described in the case study could also be conducted by varying the initial $Pset$ and the test adequacy criterion. 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 12 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. 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. 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 Discretionary Access Control (DAC) and MAC [27], we cannot generalize the results of our case study to implementations of such protocols.

Construct validity. 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], [10]. 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 are available [19], we do not know what fault in code led to these. Further, these vulnerabilities are not specific to RBAC.

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 their value would have been the same for all three procedures. Random test generation is much less costly as compared to Complete FSM-based test generation. Thus, by including the cost of test generation in the comparison of the procedures, the CBR of CRTS procedure is likely to further reduce.

9 Related Work

Ferraiolo and Kuhn [13] proposed RBAC. RBAC has been extended in [21] to include contextual and temporal constraints. Although some research has been reported in the verification of RBAC and related policies [1], [12], [23], [17], 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 DAC [7]. 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 [8], [16] use an 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 which 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 which occurs 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. [15] consider a projected state machine of the original FSM from which tests are generated using coverage criterion and test constraints. Heuristic H6 in Section 6.2 is similar to the projected state machine concept used in [15]. Ip and Dill [20] 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 [9].

10 Summary and Conclusions

A functional test generation technique that uses one of the four conformance testing procedures is proposed and evaluated. The technique raises the task of testing RBAC implementations from an ad hoc to a formal level to ease automation. 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 nonreduced model of the system. Procedure D is based on combination of test generation strategies proposed in Procedures B and C.

An empirical evaluation was carried out to assess the cost, fault detection effectiveness, and cost-benefit ratio associated with the usage of the four 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. Procedure D, based on combining the test suites generated using Procedures B and C, illustrated that the fault detection effectiveness of combined test suite increases for most of the cases.

Acknowledgments

The authors thank the anonymous reviewers for their constructive comments and suggestions.

References

[1] T. Ahmed and A.R. Tripathi, "Static Verification of Security Requirements in Role Based CSCW Systems," *Proc. Symp. Access Control Models and Technologies,* pp. 196-203, 2003.

[2] G-J. Ahn and R. Sandhu, "Role-Based Authorization Constraints Specification," *ACM Trans. Information and System Security,* vol. 3, no. 4, pp. 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 Trans. Comm.,* vol. 39, no. 11, pp. 1604-1615, Nov. 1991.

[4] J.H. Andrews, L.C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" *Proc. Int'l Conf. Software Eng.,* pp. 402-411, 2005.

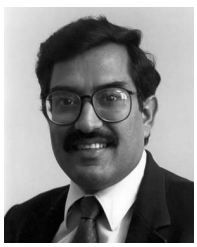
[5] F. Belli and R. Crisan, "Towards Automation of Checklist-Based Code Reviews," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 24-33, 1996.

[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. Information and System Security,* vol. 8, no. 2, pp. 187-227, 2005.

[7] R. Chandramouli and M. Blackburn, "Automated Testing of Security Functions Using a Combined Model & Interface Driven Approach," *Proc. 37th Hawaii Int'l Conf. System Sciences,* pp. 299-308, 2004.
[8] T.S. Chow, "Testing Software Design Modelled by Finite State Machines," *IEEE Trans. Software Eng.,* vol. 4, no. 3, pp. 178-187, May 1978.
[9] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," *IEEE Software,* vol. 13, no. 5, pp. 83-89, Sept. 1996.
[10] M. Daran and P. Thèvenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 158-171, 1996.
[11] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection," *Computer,* vol. 11, no. 4, pp. 34-41, Apr. 1978.
[12] M. Drouineaud, M. Bortin, P. Torrini, and K. Sohr, "A First Step Towards Formal Verification of Security Policy Properties for RBAC," *Proc. Int'l Conf. Quality Software,* pp. 60-67, 2004.
[13] D. Ferraiolo and R. Kuhn, "Role-Based Access Control," *Proc. NIST-NSA Computer Security Conf.,* pp. 554-563, 1992.
[14] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, and R. Chandramouli, "Proposed NIST Standard for Role-Based Access Control," *ACM Trans. Information and System Security,* vol. 4, no. 3, pp. 224-274, 2001.
[15] G. Friedman, A. Hartman, K. Nagin, and T. Shiran, "Projected State Machine Coverage for Software Testing," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 134-143, 2002.
[16] S. Fujiwara, G.V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test Selection Based on Finite State Models," *IEEE Trans. Software Eng.,* vol. 17, no. 6, pp. 591-603, June 1991.
[17] F. Hansen and V. Oleshchuk, "Conformance Checking of RBAC Policy and Its Implementation," *Proc. Information Security Practice and Experience Conf.,* R.H. Deng, F. Bao, H-H. Pang, and J. Zhou, eds., 2005.
[18] *ANSI RBAC Standard,* http://ite.gmu.edu/list/journals/tissec/ANSI+INCITS+359 2004.pdf, 2008.
[19] *Common Vulnerabilities and Exposures,* http://www.cve.mitre.org/, 2009.
[20] C.N. Ip and D.L. Dill, "Better Verification through Symmetry," *Formal Methods System Design,* vol. 9, nos. 1/2, pp. 41-75, 1996.
[21] J.B.D. Joshi, E. Bertino, U. Latif, and A. Ghafoor, "A Generalized Temporal Role-Based Access Control Model," *IEEE Trans. Knowledge and Data Eng.,* vol. 17, no. 1, pp. 4-23, Jan. 2005.
[22] J.B.D. Joshi, B. Shafiq, A. Ghafoor, and E. Bertino, "Dependencies and Separation of Duty Constraints in GTRBAC," *Proc. Symp. Access Control Models and Technologies,* pp. 51-64, 2003.
[23] E.C. Lupu and M. Sloman, "Conflicts in Policy-Based Distributed Systems Management," *IEEE Trans. Software Eng.,* vol. 25, no. 6, pp. 852-869, Nov./Dec. 1999.
[24] Y-S. Ma, J. Offutt, and Y-R. Kwon, "MuJava: An Automated Class Mutation System," *Software Testing, Verification and Reliability,* vol. 15, no. 2, pp. 97-133, 2005.
[25] A. Masood, R. Bhatti, A. Ghafoor, and A. Mathur, "Scalable and Effective Test Generation for Role Based Access Control Systems," Technical Report TR 2006-24, Center for Education and Research in Information Assurance and Security (CERIAS), Purdue Univ., 2006.
[26] K.K. Sabnani and A.T. Dahbura, "A Protocol Test Generation Procedure," *Computer Networks and ISDN Systems,* vol. 15, pp. 285-297, 1988.
[27] R. Sandhu and P. Samarati, "Access Control: Principles and Practice," *IEEE Comm.,* vol. 32, no. 9, pp. 40-48, Sept. 1994.
[28] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman, "Role-Based Access Control Models," *Computer,* vol. 29, no. 2, pp. 38-47, Feb. 1996.
[29] D.P. Sidhu and T.K. Leung, "Formal Methods for Protocol Testing: A Detailed Study," *IEEE Trans. Software Eng.,* vol. 15, no. 4, pp. 413-426, Apr. 1989.
[30] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys,* vol. 29, no. 4, pp. 366-427, Dec. 1997.

Ammar Masood recieved the PhD degree from the School of Electrical and Computer Engineering at Purdue University, West Lafayette, Indiana. He is currently working as an assistant professor in the Department of Avionics Engineering at the Institute of Avionics and Aeronautics, Air University, Islamabad, Pakistan. His research interests include information system security, software security testing and verification, access control systems, multimedia systems, and networking.

Rafae Bhatti received the PhD degree in computer engineering from Purdue University in 2006. He has been a postdoctoral researcher at the IBM Almaden Research Center and currently works at Oracle. This work was done when he was a PhD candidate in the School of Electrical and Computer Engineering at Purdue University, where he was also affiliated with the Center for Education and Research in Information Assurance and Security (CERIAS). His research interests include information systems security, with emphasis on design and administration of access management policies in distributed systems. He is a member of the IEEE.



Arif Ghafoor is currently a professor in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, Indiana, and is the director of Distributed Multimedia Systems Laboratory. He has been actively engaged in research on multimedia information systems, database security, and parallel and distributed computing. He has coedited a book entitled *Multimedia Document Systems in Perspectives* and has coauthored a book entitled *Semantic Models for Multimedia Database Searching and Browsing*. He received the IEEE Computer Society 2000 Technical Achievement Award for his research contributions in the area of multimedia systems. He is a fellow of the IEEE and a member of the IEEE Computer Society.

Aditya Mathur performs research which spans software testing, reliability, process control, and a study of the function of the human brain. His published work relates to investigations into the effectiveness of testing techniques and their applicability to the testing of sequential and distributed software, methods for the estimation of software reliability, and techniques and tools for managing a collection of Internet-enabled devices. He is the author of several textbooks, including the recently published *Foundations of Software Testing*. He is a member of the IEEE Computer Society.