A Framework for Enforcing Constrained RBAC Policies

Jason Crampton
Information Security Group
Royal Holloway, University of London
jason.crampton@rhul.ac.uk

Hemanth Khambhammettu
Information Security Group
Royal Holloway, University of London
h.khambhammettu@rhul.ac.uk

***Abstract*—Constraints are an important part of role-based access control policies. The safety or security of a system is maintained by enforcing constraints that are specified in the policy. In order to decide whether an access request is authorized, existing constraint enforcement mechanisms perform both *authorization checking*, which verifies that the requested operation is sufficiently authorized, and *constraint checking*, which checks whether permitting the operation would violate any constraint. The decision functions of large-scale systems, where hundreds of requests arise concurrently, require relatively simple decision-making algorithms. Performing constraint checking when deciding whether an access request is authorized introduces an additional overhead.**

In this paper, we describe a new framework for enforcing constraints that *only* requires us to perform authorization checking when deciding an access request. Essentially, we transform the constraint checking problem into an authorization checking problem by modifying authorization state following the success of an access request. We present a framework for enforcing constrained role-based access control policies, and describe relevant algorithms for updating the authorization state.

I. Introduction

An *access control policy* specifies high-level rules according to which the system must govern access to its protected resources. An *access control mechanism* defines low level functions that implement the controls imposed by the policy. Typically, an access control mechanism includes a *reference monitor*, which mediates every request and decides whether to *permit* or *deny* the request.

Often, the success of an access request not only allows a principal to access protected resources, but may also update the authorization state. In a role-based policy, for example, the authorization state is updated following the success of a request that attempts to assign a user to a role. In this paper, we consider access requests that require the authorization state to be updated following their success. We refer to such an access request as an *operation*.

Constraints specify high-level policy requirements that must be satisfied in order to enforce the business rules of the system. A "constrained policy" includes the specification of at least one constraint on the set of operations that may be permitted by the system. In order to guarantee that the constrained policy is enforced, we must ensure that no operation which results in the violation of a constraint can succeed.

Given a constrained policy, in order to decide whether an operation is to be permitted, the reference monitor must perform *authorization checking* and *constraint checking*.

- Authorization checking is required to verify that the initiator of the operation has the (minimal set of) permissions required for the operation to be permitted.
- Constraint checking is required to ensure that permitting the given operation will not result in a state change that would cause the violation of any constraint specified in the policy. (In other words, constraint checking is essential for enforcing a constraint.)

An operation is only permitted if it passes both authorization checking and constraint checking.

Generally, in order to know whether or not permitting an operation *op* would violate any constraint, it is necessary to identify a set of constraints that are relevant to *op* and evaluate these constraints at some point in time. Existing constraint enforcement mechanisms perform constraint checking while deciding an operation [1]–[3]. We believe that approaches that perform both authorization checking and constraint evaluation for deciding an operation may not be appropriate for applications that require quick access decision times. This is because performing constraint checking for deciding an operation creates an additional overhead to access decision times.

The decision functions of large-scale systems, where hundreds of requests arise concurrently, require relatively simple decision-making algorithms. We describe an approach where an operation is decided by only performing authorization checking. In simple terms, we transform the constraint checking problem into an authorization checking problem by updating the authorization state following the authorization of an access request. The authorization state is updated (after the request has been authorized) to include those requests that are now prohibited as a result of permitting the authorized request to proceed. The evaluation of an access request first checks that the request is not prohibited, and only then decides whether it is authorized.

Hence our framework has the following features.

- Authorization state encodes both *positive* and *negative* authorizations.
- Constraints are *not* evaluated when deciding an operation;

rather, constraints are evaluated following the success of an operation.

A variety of constraints have been considered in the role-based access control (RBAC) literature. Perhaps the best known being static separation of duty, dynamic separation of duty and cardinality constraints. We develop a uniform syntax for specifying such constraints and also illustrate that these constraints only differ in their *evaluation context*, enabling us to have a single strategy for enforcing many different types of constraints.

However, existing RBAC models do not provide a sufficiently rich notion of authorization state. The concept of a *session*, for example, is defined simply as a set of roles selected by a user. In order to enforce certain types of dynamic constraint it is necessary to have a more complex data structure to describe sessions. The first part of this paper, therefore, describes some minimal extensions to the basic RBAC model that are required to support the enforcement of many types of constraints.

The following are the main contributions of this paper.

- We extend the RBAC96 model to include a comprehensive account of run-time and historic authorization state and define state changes within that model.
- We define a uniform syntax for specifying both separation of duty and cardinality constraints.
- We define formal semantics for constraint satisfaction.
- We present a framework for enforcing constrained role-based access control policies, and describe relevant algorithms for updating both positive and negative authorization state.

The rest of the paper is organized as follows. In the next section, we extend the RBAC96 model. Section III formally defines constraints, evaluation contexts and semantics for constraint satisfaction. We describe a new framework for enforcing constraints in Section IV. We compare our work with relevant work of the literature in Section V, and conclude this work in Section VI.

II. RBAC MODEL

We extend the RBAC96 model [4] which defines the following sets and relations: R is a countable set of roles, U is a countable set of users, P is a countable a set of permissions, and S is a countable set of sessions. We retain the user-role and permission-role assignment relations, $UA \subseteq U \times R$ and $PA \subseteq P \times R$.

A. Extensions to RBAC96

However, we extend the accepted notion of *session* and include additional relations, because the RBAC96 model is not sufficiently rich semantically to permit the evaluation of certain types of constraints. RBAC96 defines a set of sessions S, and functions $user : S \to U$ and $roles : S \to 2^R$, where $user(s)$ denotes the user who activated session s and $roles(s)$ denotes the set of roles active in session s; $roles(s)$ is a subset of the roles for which $user(s)$ is authorized. While these elements of the model permit the evaluation of constraints concerned with the current state of the system (albeit in a rather complicated way), they do not permit the evaluation of constraints concerned with past actions. Accordingly, we introduce the following relations.

- The user-session relation $US \subseteq U \times S \times \{0,1\}$.
  - $(u,s,1) \in US$ means that session s is associated with user u and is live.
  - $(u,s,0) \in US$ means that s is no longer live (having been terminated by the user).

  We assume that each session s is associated with a unique identifier. In particular, once a session is terminated it is never reactivated. Note, however, that a user may have two or more live sessions: that is, we may have $(u,s_1,1) \in US$ and $(u,s_2,1) \in US$.
  In other words, a session identifier uniquely identifies a tuple in US, but the user attribute does not.
- The session-role relation $SR \subseteq S \times R \times \{0,1\}$.[1]
  - $(s,r,1) \in SR$ means that role r is currently active in session s.
  - $(s,r,0) \in SR$ means that role r is no longer active in session s.

  Hence the set of roles activated by a user u in session s can be computed from SR. Moreover, the set of all roles activated by a user u in all live sessions can be computed from US and SR.
- The session-permission relation $SP \subseteq S \times P \times \{0,1\}$.[2] This relation is used to model the actual granting of permissions in response to requests from a session. When the permission is no longer required, we say that it is *released*.
  - $(s,p,1) \in SP$ means that session s has requested and been granted permission p, and p has not yet been released by s. The authorization semantics of RBAC means that there exists $r \in R$ such that $(s,r,1) \in SR$ and $(p,r) \in PA$.
  - $(s,p,0) \in SP$ means that session s requested p, was granted permission p, and has subsequently released it.

B. State of an RBAC system

The (*authorization*) *state* of a role-based system may be divided into three parts.

- The *static* authorization state λ_s defines authorizations for various entities, such as users and roles, within a system.
- The *runtime* or *dynamic* authorization state λ_d defines what roles are currently active and which permissions

[1] The RBAC96 model permits the set of roles within a session to be dynamic, but the specification of the model only allows us to identify the roles that are currently active within a session. This is inadequate if we wish to enforce dynamic constraints that restrict activation of roles across multiple sessions.

[2] Note also that the RBAC96 model admits the existence of a set of active permissions within a session. Again, we need to extend the specification of the RBAC96 model so that we can enforce dynamic constraints that restrict invocation of permissions across multiple sessions.

have been granted (as a consequence of an access request being authorized), but not released.

- The *historic* authorization state λ_h defines previous system activity, such as successful activation of roles and invocation of permissions.

The static authorization state is represented using existing RBAC96 structures, such as the UA and PA relations. In particular, $\lambda_s = UA \cup PA$. Note that the extensions defined in Section II-A can be used to model the run-time state of the system, as well as the historic state. This is because we include the "active flag" in such relations, enabling us to distinguish between current role activations and permission invocations and those that have happened in sessions that are no longer live. Hence, $\lambda_d \subseteq \lambda_h = US \cup SR \cup SP$.

C. State changes

An event is triggered by the successful completion of some operation. Such an operation may be initiated either *internally* by the system or *externally* by a user (in the form of an access request, for example).

Each event is associated with a set of actions. Such an action may either simply update the state or perform some more complex combination of tasks. In the context of this paper, each such action has the effect of modifying some RBAC relation defined in Section II-A. In other words a successful operation causes a state change, while an unsuccessful operation does not. The successful completion of a user-role assignment operation, for example, causes a change in the UA relation; whereas a successful activation of a role by a session changes the SR relation.

Table I summarizes the operations that we consider in this paper. It includes the parameters associated with each operation (where $u \in U$, $r \in R$, $s \in S$ and $p \in P$), the state that is affected, and the effect of the state change.

Any operation that causes a tuple to be added to the state is said to be a *grant* operation; and any operation that causes a tuple to be removed from the state is said to be a *revoke* operation. Hence, $invokePerm(.)$ is a grant operation and $releasePerm(.)$ is a revoke operation, for example.

III. Constraints: Specification and Evaluation

Informally, constraints encode business requirements, which specify restrictions on the state of a system. Generally, constraints are used to specify those states that should be prohibited.

Several kinds of constraints have previously been identified for RBAC models [3], [5]–[8]. Such constraints are generally classified in the following way.

- A *static* constraint specifies restrictions on the components of authorization state.
  For example, we may specify a static constraint that specifies no user is assigned to both roles r_1 and r_2. That is, $\{(u, r_1), (u, r_2)\} \nsubseteq UA$.
- A *dynamic* constraint specifies restrictions on the components of runtime state.
  For example, we may specify a dynamic constraint that specifies no session may simultaneously activate both roles r_1 and r_2 within a session. That is, $\{(s, r_1, 1), (s, r_2, 1)\} \nsubseteq SR$.
  We may also specify a dynamic constraint that specifies no user may simultaneously activate both roles r_1 and r_2 across sessions. That is, for any two sessions $s_1, s_2 \in S$, such that $(u, s_1, 1), (u, s_2, 1) \in US$, $\{(s_1, r_1, 1), (s_2, r_2, 1)\} \nsubseteq SR$ and $\{(s_1, r_2, 1), (s_2, r_1, 1)\} \nsubseteq SR$.
- A *historic* constraint specifies restrictions on the components of historic state.
  For example, we may specify a historic constraint that specifies no user may ever invoke both permissions p_1 and p_2. That is, for any two sessions $s_1, s_2 \in S$, such that $(u, s_1, b_1), (u, s_2, b_2) \in US$, $\{(s_1, p_1, b_1'), (s_2, p_2, b_2')\} \nsubseteq SP$ and $\{(s_1, p_2, b_1''), (s_2, p_1, b_2'')\} \nsubseteq SP$.

The literature suggests that there is little consensus within the research community on specifying historic constraints that restrict invocation of permissions by roles [6]. Moreover, it is sessions (that is, sets of roles) that invoke permissions, rather than individual roles. Hence, we do not believe it is appropriate to consider constraints that restrict invocation of permissions by single roles.

Crampton proposed a simple specification model for defining separation of duty constraints [6]. In this section, we extend this model to include both separation of duty and cardinality constraints for RBAC models.

Definition 1: A *constraint* is a tuple $(\mathcal{D}, \mathcal{S}, k, x)$ where $\mathcal{D}$ is the *domain* of the constraint, $\mathcal{S}$ is the *constraint set*, k is the *threshold* and x is the *state context* of the constraint.

Given a constraint $c = (\mathcal{D}, \mathcal{S}, k, x)$, the domain $\mathcal{D}$ and the constraint set $\mathcal{S}$ are subsets of one of U, R, P and S. The context of a constraint x specifies an environment, such as *static*, *dynamic* and *historic*, within which the constraint is to be evaluated. For example, a static state context requires that the constraint be evaluated with reference to (part of) the static authorization state (λ_s).

Example 2: A constraint $c_1 = (U, \{r_1, r_2\}, 1, s)$ specifies that a user $u \in U$ may be assigned to no more than one role from the specified set of roles $\{r_1, r_2\}$.

That is, c_1 represents a simple separation of duty constraint for assignment of a user to roles r_1 and r_2.

Example 3: A constraint $c_2 = (S, \{r_1, r_2, r_3\}, 2, d)$ specifies that a session $s \in S$ may activate no more than two roles from the specified set of roles $\{r_1, r_2, r_3\}$.

That is, c_2 represents a cardinality constraint that limits the number of roles in $\{r_1, r_2, r_3\}$ that may be activated within a session.

In the remainder of this paper, we will use the constraints defined as running examples to illustrate constraint evaluation and enforcement.

A. Evaluation context

As mentioned earlier, constraints are evaluated by referring to some part of the authorization state. However, the relevant

TABLE I
EVENTS AND THEIR ASSOCIATED ACTIONS

Operation	Parameters	Affected state	Action
$assignUser$	u, r	λ_s	$UA \leftarrow UA \cup \{(u,r)\}$
$revokeUser$	u, r	λ_s	$UA \leftarrow UA \setminus \{(u,r)\}$
$assignPerm$	p, r	λ_s	$PA \leftarrow PA \cup \{(p,r)\}$
$revokePerm$	p, r	λ_s	$PA \leftarrow PA \setminus \{(p,r)\}$
$createSession$	u, s	λ_d, λ_h	$US \leftarrow US \cup \{(u,s,1)\}$
$destroySession$	u, s	λ_d, λ_h	$US \leftarrow (US \cup \{(u,s,0)\}) \setminus \{(u,s,1)\}$
$activateRole$	s, r	λ_d, λ_h	$SR \leftarrow SR \cup \{(s,r,1)\}$
$deactivateRole$	s, r	λ_d, λ_h	$SR \leftarrow (SR \cup \{(s,r,0)\}) \setminus \{(s,r,1)\}$
$invokePerm$	s, p	λ_d, λ_h	$SP \leftarrow SP \cup \{(s,p,1)\}$
$releasePerm$	s, p	λ_d, λ_h	$SP \leftarrow (SP \cup \{(s,p,0)\}) \setminus \{(s,p,1)\}$

part is determined by the domain, constraint set and state context of a constraint.

Definition 4: Let $(\mathcal{D}, \mathcal{S}, k, x)$ be a constraint. Then, the *evaluation context* of $(\mathcal{D}, \mathcal{S}, k, x)$ is defined by the combination of $\mathcal{D}$, $\mathcal{S}$ and x. We write $\mathcal{E}(\mathcal{D}, \mathcal{S}, x)$ to denote the evaluation context of $(\mathcal{D}, \mathcal{S}, k, x)$.

A constraint is evaluated by referring to its corresponding evaluation context.[3] If $\mathcal{D} \subseteq U$, $\mathcal{S} \subseteq R$ and $x = s$, as in Example 2, then $\mathcal{E}(U, R, s) = UA \subseteq \lambda_s$. Hence, a constraint (U, R, k, s) is evaluated by referring to the UA relation (part of the static authorization state λ_s).

Note that different combinations of the domain, constraint set and context give rise to different constraints and evaluation contexts. Specifically, there are at most $4 \times 3 \times 3 = 36$ different constraint types that can arise, and 18 different evaluation contexts since $\mathcal{E}(\mathcal{D}, \mathcal{S}, x) = \mathcal{E}(\mathcal{S}, \mathcal{D}, x)$. Some of these constraints and evaluation contexts will be more useful than others, while some are not applicable. In particular, the SR relation is likely to be widely used as an evaluation context because it is used to determine whether dynamic separation of duty and cardinality constraints on role activation within a session are satisfied.

Table II shows how a number of evaluation contexts may be derived from the basic relations we defined, in Section II-A, for representing authorization state. We specify the derivation using relational algebra, where $\pi_{A_1,\ldots,A_n}(R)$ denotes the projection of relation R on attributes $A_1, \ldots, A_n$, $\sigma_{exp}(R)$ denotes the selection of those tuples in R that satisfy the logical expression exp, and $\bowtie$ denotes the join of two relations. These evaluation contexts are used when evaluating various constraints. (The relations UA, PA, US, SP and SR are evaluation contexts that need no derivation from other relations.) There is a very natural correspondence between constraints and their respective evaluation contexts, a selection of which are shown in Table III.

TABLE II
DERIVED EVALUATION CONTEXTS

Relation	Derivation	Semantics
$UP_s \subseteq U \times P$	$\pi_{U,P}(UA \bowtie PA)$	authorized user-permission relationships
$US_d \subseteq U \times S$	$\sigma_{Active=1}(US)$	current user-session activations
$SR_d \subseteq S \times R$	$\sigma_{Active=1}(SR)$	current session-role activations
$UR_d \subseteq U \times R$	$\pi_{U,P}(US_d \bowtie SR_d)$	current user-role activations
$SP_d \subseteq S \times P$	$\sigma_{Active=1}(SP)$	current session-permission invocations
$UP_d \subseteq U \times P$	$\pi_{U,P}(US_d \bowtie SP_d)$	current user-permission invocations
$UR_h \subseteq U \times R$	$\pi_{U,P}(US \bowtie SR)$	all user-role activations
$UP_h \subseteq U \times P$	$\pi_{U,P}(US \bowtie SP)$	all user-permission invocations

TABLE III
EXAMPLES OF CONSTRAINTS AND THEIR EVALUATION CONTEXTS

Constraint			Evaluation context
Domain	Constraint set	Context	
U	R	*static*	UA
R	U	*static*	UA
P	R	*static*	PA
U	S	*dynamic*	US_d
S	R	*dynamic*	SR_d
U	R	*dynamic*	UR_d
S	P	*dynamic*	SP_d
U	P	*dynamic*	UP_d
S	P	*historic*	SP_h
U	P	*historic*	UP_h

B. Constraint violation

A constraint defines a family of sets and specifies a maximum number of elements to be permitted in each set in the specified context. The constraint $c = (A, B, k, x)$ defines the family of sets

$$\begin{aligned}\mathcal{Q}_c &= \bigcup_{a_i \in A} (\{a_i\} \times B) \\ &= \{\{(a_1,b_1), \ldots, (a_1,b_n)\}, \ldots, \{(a_m,b_1), \ldots, (a_m,b_n)\}\}.\end{aligned}$$

For each $a_i \in A$, let $Q_c^{a_i}$ denote $\{(a_i,b_1), \ldots, (a_i,b_n)\}$. Then, $Q_c^{a_i} \in \mathcal{Q}_c$ is called a *constrained (authorization) set* and $q \in Q_c^{a_i}$ is called a *constrained (authorization) request*. Note that $Q_c^{a_i} \subseteq \mathcal{E}(A, B, x)$.

[3]Note that, at any given point in time, an evaluation context is *always* a subset of the authorization state.

Definition 5: A constraint $c = (A, B, k, x)$ is *violated* if, for any $a \in A$, $|Q_c^a \cap \mathcal{E}(A, B, x)| > k$.

Informally, a constraint (A, B, k, x) is *violated* if its evaluation context contains more than k elements from any one of its constrained sets.

Definition 6: A constraint $c = (A, B, k, x)$ is said to be *violation-prone* if, for any $a \in A$, $|Q_c^a \cap \mathcal{E}(A, B, x)| = k$.

In other words, a constraint is violation-prone if a subsequent state change could cause its violation. If we wish to prevent the violation of constraints, then it is the set of violation-prone constraints that are of concern at any given time.

IV. Preventing Constraint Violation

Existing approaches to constraint enforcement perform both authorization checking and constraint checking while deciding a constrained operation [1]–[3]. A detailed discussion of such constraint enforcement mechanisms is given in Section IV-D. We believe that such approaches incur higher access decision times, which is due to the fact that constraint checking is performed while deciding the request.

In this section, we describe an alternative approach for enforcing constraints that only requires us to perform authorization checking. Essentially, we transform the constraint checking problem into an authorization checking problem by updating the authorization state.

Specifically, given the success of an operation, we appropriately update authorization policy and evaluate relevant constraints for determining a set of "prohibited" operations that would violate at least one constraint. This prohibited set of operations is also included as part of the authorization policy. The reference monitor denies every subsequent operation that is prohibited by the policy.

A. Authorization state

Recall, from Section II-B, that the authorization state encodes authorizations that have been permitted. In this section, we extend this authorization state to encode both *permitted* (positive) and *prohibited* (negative) authorizations.

We represent the *extended* authorization state A by the following two relations.

- $A^+ \subseteq X \times Y$: a tuple $(x, y) \in A^+$ means that an operation that authorizes x for y has been permitted. We refer to A^+ as the *authorization relation*.
- $A^- \subseteq X \times Y$: a tuple $(x, y) \in A^-$ means that an operation that authorizes x for y has been prohibited (because authorizing x for y would violate at least one constraint). We refer to A^- as the *constraint enforcement relation*.

Recall, from Section III-A, that we defined a suite of evaluation contexts for evaluating constraints, which are derived from authorization state. Hence, these evaluation contexts naturally encode successful positive authorizations and are analogous to the authorization relation (A^+) defined above. In the following, we will use a superscript '+' for all evaluation contexts that encode successful positive authorizations. The evaluation context UA that encodes successful user-role assignments, for example, is now denoted as UA^+.

We define a constraint enforcement relation for each evaluation context. Table IV shows constraint enforcement relations with their semantics.

TABLE IV
Constraint enforcement relations and their semantics

Constraint enforcement relation	**Semantics**
$UA_s^- \subseteq U \times R$	$(u, r) \in UA_s^-$ means u can not be assigned to r
$PA_s^- \subseteq P \times R$	$(p, r) \in PA_s^-$ means p can not be assigned to r
$UP_s^- \subseteq U \times P$	$(u, p) \in UP_s^-$ means u can not be authorized to p
$SR_d^- \subseteq S \times R$	$(s, r) \in SR_d^-$ means r can not be activated in s
$UR_d^- \subseteq U \times R$	$(u, r) \in UR_d^-$ means u can not activate r
$SP_d^- \subseteq S \times P$	$(s, p) \in SP_d^-$ means p can not be invoked in s
$UP_d^- \subseteq U \times P$	$(u, p) \in UP_d^-$ means u can not invoke p in any session
$UR_h^- \subseteq U \times R$	$(u, r) \in UR_h^-$ means u can no longer activate r
$UP_h^- \subseteq U \times P$	$(u, p) \in UP_h^-$ means u can no longer invoke p

Then, the extended authorization state of an RBAC system is defined as follows.

- $\lambda_s = UA^+ \cup UA^- \cup PA^+ \cup PA^-$;
- $\lambda_d = US^+ \cup SR_d^+ \cup SR_d^- \cup UR_d^- \cup SP_d^+ \cup SP_d^- \cup UP_d^-$;
- $\lambda_h = US^+ \cup SR_h^+ \cup SR_h^- UR_h^- \cup SP_h^+ \cup SP_h^- \cup UP_h^-$.

B. Access request evaluation

An operation is decided by the reference monitor by referring to the constraint enforcement relation A^- and authorization relation A^+. Specifically, an operation *op* that authorizes x for y is

- *denied* if $(x, y) \in A^-$;
- *permitted* only if $(x, y) \notin A^-$ and $(x, y) \in A^+$.

For example, a request to activate a role r in a session s created by a user u is *denied* if $(s, r) \in SR_d^-$, and *permitted* only if $(s, r) \notin SR_d^-$ and $(u, r) \in UA^+$.

C. Authorization state changes

Recall, from Section II-C, that the success of an operation triggers an event, which requires the corresponding set of actions to be executed. In order to enforce constrained policies, we require that such a set of actions update both *positive* and *negative* authorization state. The precise updates, however, are dependent on whether the successful operation is a grant or revoke operation.

1) Grant operations: A successful grant operation causes a tuple to be added to some component of the authorization state. This may result in some constraints becoming violation-prone (that is, the threshold value may be attained for some constraints). Hence, we identify these violation-prone constraints, compute a set of prohibited operations and update the policy by including negative authorizations for the set of prohibited operations. In other words, we extend the state transition

model to incorporate updates to the negative authorization policy.

Specifically, the success of a grant operation *op* that authorizes x for y triggers an event E whose action set performs the following.

- $A^+ \leftarrow A^+ \cup \{(x, y)\}$ (that is, update the positive/permitted authorization policy to include positive authorization);
- compute a set of prohibited operations $\neg\mathsf{Q}$ by evaluating violation-prone constraints;
- $A^- \leftarrow A^- \cup \neg\mathsf{Q}$ (that is, update the negative/prohibited authorization policy to include negative authorizations).

Given the success of a grant operation and a set of constraints C that are specified in the policy, the following three functions must be performed for determining a set of prohibited operations $\neg\mathsf{Q}$.

1) identify the set of constraints $C \subseteq \mathsf{C}$ that restrict the successful grant operation;
2) determine a set of constraints $C' \subseteq C$ that have become violation-prone (that is, determine constraints for which the threshold has been attained);
3) compute the set of prohibited operations for each violation-prone constraint $c' \in C'$.

Figure 1 illustrates pseudo-code algorithms that implement the above functions. We now describe these pseudo-code algorithms. Figure 1(a) illustrates a simple algorithm for a sub-routine that takes parameters of the successful grant operation $\{x, y\}$ and the set of constraints C as inputs and returns the set of constraints that restrict the successful operation. Essentially, for each constraint $c \in \mathsf{C}$, the algorithm checks whether a tuple formed by the set of operation parameters x and y belongs to the constrained set of c (steps 02-03). In other words, the tuples formed by the operation parameters identify the set of relevant constraints. The algorithm returns a set of constraints C that restrict operation with parameters x and y (step 04).

Figure 1(b) illustrates another simple algorithm for another sub-routine that takes parameters of the successful grant operation $\{x, y\}$, the evaluation context EC^+ and the set of constraints C returned by algorithm `affectedConstraints()` as inputs and returns the set of violation-prone constraints. Essentially, for each constraint $c \in C$, this algorithm checks whether c has become violation-prone (steps 02-04). In other words, the algorithm checks whether the threshold has been attained for c. The algorithm returns a set of violation-prone constraints C' (step 05).

Figure 1(c) illustrates an algorithm for another sub-routine that takes parameters of the successful grant operation $\{x, y\}$, the evaluation context EC^+ and the set of constraints C as inputs and returns the set of prohibited operations. Firstly, the algorithm determines a set of constraints C that restrict the successful grant operation by invoking the sub-routine `affectedConstraints(`$\{x, y\}$`,`C`)` (Figure 1(a), step 02). Secondly, the algorithm determines a set of violation-prone constraints C' by invoking the sub-routine `violation-proneConstraints(`EC^+, C`)` (Figure 1(b), step 03). Subsequently, for each violation-prone constraint $c' \in C'$, the algorithm determines a relevant constrained set and computes a set of operations $\neg Q'_c$ that violate c' (steps 05-06). The algorithm appends the set of operations $\neg Q'_c$ to the cumulative set of prohibited operations $\neg\mathsf{Q}$ (step 07). Finally, the algorithm returns the set of prohibited operations $\neg\mathsf{Q}$ (step 08).

```
Inputs: grant operation parameters x,y
        set of constraints C

Output: set of constraints that restrict q

01 let C = ∅
02 for each constraint c ∈ C
03   if (((x,y) ∈ Q_c^x) ∨ ((y,x) ∪ Q_c^y)) then
04     set C ← C ∪ {c}
04 return C
```

(a) Algorithm `affectedConstraints()`

```
Inputs: grant operation parameters x,y
        evaluation context EC+
        set of affected constraints C

Output: set of violation-prone constraints

01 let C' = ∅
02 for each constraint c ∈ C
03   if ((|Q_c^x ∩ EC+| = k_c) ∨ (|Q_c^y ∩ EC+| = k_c)) then
04     set C' ← C' ∪ {c}
05 return C'
```

(b) Algorithm `violation-proneConstraints()`

```
Inputs: grant operation parameters x,y
        evaluation context EC+
        set of constraints C

Output: set of prohibited operations

01 let ¬Q = ∅
02 let C = affectedConstraints({x,y},C)
03 let C' = violation-proneConstraints(EC+,C)
04 for each constraint c' ∈ C'
05     if ((x,y) ∈ Q_c'^x) then let ¬Q_c' = Q_c'^x \ EC+
06     else let ¬Q_c' = Q_c'^y \ EC+
07     set ¬Q ← ¬Q ∪ ¬Q_c'
08 return ¬Q
```

(c) Algorithm `computeProhibitedSet()`

Fig. 1. Determining prohibited operations

Consider, for example, the constraint $c_2 = (S, \{r_1, r_2, r_3\}, 2, d)$, defined in Example 3, and assume that $(s, r_1), (s, r_2), (s, r_3) \notin SR_d$. Then, an operation that activates r_1 in s, for example, succeeds. Hence, we update $SR_d^+ \leftarrow SR_d^+ \cup \{(s, r_1)\}$ and the constraint checking process is triggered for this policy update by invoking `computeProhibitedSet(`$(s, r_1), SR_d^+, \mathsf{C}$`)`. Since the threshold of constraint c_2 is not attained at this point, the `computeProhibitedSet(.)` returns `null`, and no updates are made to the SR_d^- relation. Should a subsequent operation that activates r_2 in s succeed, we update $SR_d^+ \leftarrow SR_d^+ \cup \{(s, r_2)\}$ and the constraint checking process is triggered by invoking `computeProhibitedSet(`$(s, r_2), SR_d^+, \mathsf{C}$`)`. At this point, it is known that the threshold for the constrained set $Q_{c_2}^s$ has been attained and that c_2 has become violation-prone (for session s). Hence, the activation of role r_3

in s is prohibited, and `computeProhibitedSet(.)` returns (s, r_3). The negative authorization policy is updated to reflect this prohibited operation (returned by `computeProhibitedSet(.)`). In our example, $SR_d^- \leftarrow SR_d^- \cup \{(s, r_3)\}$.

2) Revocation operations: A successful revoke operation triggers an event E', which deletes appropriate tuples from some components of the authorization state. As a result, some constraints may no longer be violation-prone (that is, the threshold values for some constraints may no longer be attained). Hence, we compute a set of operations Q that are no longer prohibited by such constraints and update the policy by deleting negative authorizations for operations which belong in Q.

Specifically, the success of an operation *op* that revokes the authorization of x for y triggers an event E' whose action set performs the following.

- $A^+ \leftarrow A^+ \setminus \{(x, y)\}$ (that is, update the positive/permitted authorization policy to delete the authorization revoked by successful operation);
- compute a set of operations Q that are no longer prohibited by evaluating constraints that are no longer violation-prone;
- $A^- \leftarrow A^- \setminus \mathsf{Q}$ (that is, update the negative/prohibited authorization policy to delete the set of operations Q).

Given the success of a revoke operation and a set of constraints C that are specified in the policy, the following four functions must be performed for determining a set of operations Q that are no longer prohibited.

1) identify the set of constraints $C \subseteq \mathsf{C}$ that restrict the successful revocation operation;
2) determine a set of constraints $C' \subseteq C$ that are no longer violation-prone;
3) compute the set of operations $\neg\mathsf{Q}$ that were prohibited by the set of constraints C';
4) compute a set of operations $\mathsf{Q} \subseteq \neg\mathsf{Q}$ that are no longer prohibited.

Figure 2 illustrates a suite of pseudo-code algorithms for implementing the above functions. Essentially, these algorithms compute the revised authorization state following a successful revoke operation. We now describe these pseudo-code algorithms.

Figure 2(a) illustrates a simple algorithm that determines a set of constraints C' that are no longer violation-prone. This algorithm accepts parameters of the successful revoke operation a and b, the evaluation context EC^+ and the set of constraints C that restrict the revoke operation as inputs, and returns a set of safe constraints $C' \subseteq \mathsf{C}$, that are no longer violation-prone. Specifically, given a set of affected constraints C, the algorithm determines a set of constraints $C' \subseteq C$ for which the number of permitted operations, for each constraint $c' \in C'$, has reached its penultimate threshold (steps 03-04). The algorithm returns the set of constraints C' (step 05).

Figure 2(b) illustrates an algorithm that accepts parameters of the successful revoke operation a and b, an evaluation context EC^- and a set of constraints C' that are no longer violation-prone (returned by the sub-routine `affectedSafeConstraints()`) as inputs, and returns a set of operations that were previously prohibited by constraints that belong in C'. Essentially, for each constraint $c' \in C'$, the algorithm determines a set of operations $\neg Q'_c$ that were prohibited by c' (steps 03-04). The algorithm appends the set of operations $\neg Q'_c$ to the cumulative set of prohibited operations $\neg\mathsf{Q}$ (step 05). Finally, the algorithm returns the set of prohibited operations $\neg\mathsf{Q}$ that were prohibited by the set of constraints C' (step 06).

Figure 2(c) illustrates an algorithm that accepts parameters a' and b' of a prohibited operation $\neg q$, an evaluation context EC^+ and the set of constraints C as inputs, and determines whether $\neg q$ is still prohibited. The algorithm returns a boolean value: `true` means that $\neg q$ is still prohibited and `false` means that $\neg q$ is no longer prohibited. Essentially, for each constraint $c \in \mathsf{C}$, the algorithm checks whether c restricts $\neg q$ and c is violation-prone (steps 01-03). Should c restrict $\neg q$ and c is violation-prone, then this would mean that constraint c prohibits $\neg q$. In such a case, the algorithm terminates prematurely by returning `true` (steps 02-03). If no constraint prohibits $\neg q$, the algorithm returns `false` indicating that $\neg q$ is no longer prohibited (step 04).

Figure 2(d) illustrates an algorithm for determining a set of operations that are no longer prohibited following the success of a revocation operation. The algorithm accepts parameters of the successful revocation operation a and b, an evaluation context EC^+, an evaluation context EC^- and the set of constraints C as inputs, and returns a set of operations Q that are no longer prohibited. Firstly, the algorithm identifies a set of constraints $C \subseteq \mathsf{C}$ that restrict the successful revoke operation q by invoking the sub-routine `afftectedConstraints`(q, C) (Figure 1(a), step 02). Secondly, the algorithm determines a set of constraints $C' \subseteq C$ that are no longer violation-prone by invoking the sub-routine `notViol-proneConstraints`$((a, b), EC^+, C)$ (Figure 2(a), step 03). Thirdly, the algorithm computes a set of operations $\neg\mathsf{Q}$ that were previously prohibited by the set of constraints C' by invoking `prohibitedSet`$((a, b), EC^-, C')$ (Figure 2(b), step 04). Subsequently, for each prohibited operation $\neg q \in \neg\mathsf{Q}$, the algorithm checks whether $\neg q$ is still prohibited by invoking `prohibitedOperation`$((a', b'), EC^+, \mathsf{C})$ (Figure 2(c), step 06). The algorithm appends an operation that is no longer prohibited to the set of operations Q (step 07). Finally, the algorithm returns a set of operations Q that are no longer prohibited (step 08).

We illustrate revocation updates with the running example discussed in Section IV-C1. Recall that we considered the constraint $c_2 = (S, \{r_1, r_2, r_3\}, 2, d)$ and the operations `activateRole`(s, r_1) and `activateRole`(s, r_2) were successful. Hence, we have $SR_d^+ = \{(s, r_1), (s, r_2)\}$ and $SR_d^- \leftarrow SR_d^- \cup \{(s, r_3)\}$. Should a subsequent operation that deactivates role r_1 in s succeed, we update $SR_d^+ \leftarrow SR_d^+ \setminus \{(s, r_2)\}$ and trigger the constraint checking process by invoking

```
Inputs: revoke operation parameters a,b
       evaluation context EC+
       set of affected constraints C

Output: set of constraints that are no longer
violation-prone

01 let C' = ∅
02 for each constraint c ∈ C
03     if ((|Q_c^a ∩ EC+| = k_c − 1) ∨ (|Q_c^b ∩ EC+| = k_c − 1))
then
04         set C' ← C' ∪ {c}
05 return C'
```

(a) Algorithm `notViol-proneConstraints()`

```
Inputs: revoke operation parameters a,b
       evaluation context EC−
       set of affected safe constraints C'

Output: set of prohibited operations

01 let ¬Q = ∅
02 for each constraint c' ∈ C'
03     if ((a,b) ∈ Q_{c'}^a) then let ¬Q_{c'} = Q_{c'}^a ∩ EC−
04     else let ¬Q_{c'} = Q_{c'}^b ∩ EC−
05     set ¬Q ← ¬Q ∪ ¬Q_{c'}
06 return ¬Q
```

(b) Algorithm `prohibitedSet()`

```
Inputs: prohibited operation parameters a',b'
       evaluation context EC+
       set of constraints C

Output: a boolean value

01 for each constraint c ∈ C
02     if (((a',b') ∈ Q_c^{a'}) ∧ (|Q_c^{a'} ∩ EC+| = k_c)) then
return true
03     if (((b',a') ∈ Q_c^{b'}) ∧ (|Q_c^{b'} ∩ EC+| = k_c)) then
return true
04 return false
```

(c) Algorithm `prohibitedOperation()`

```
Inputs: revoke operation parameters a,b
       evaluation context EC+
       evaluation context EC−
       set of constraints C

Output: set of operations that are no longer
prohibited

01 let Q = ∅
02 let C = affectedConstraints((a,b),C)
03 let C' = affectedSafeConstraints((a,b),EC+,C)
04 let ¬Q = prohibitedSet((a,b),EC−,C')
05 for each prohibited operation (a',b') ∈ ¬Q
06     if (prohibitedOperation((a',b'),EC+,C) =
false) then
07         set Q ← Q ∪ {(a',b')}
08 return Q
```

(d) Algorithm `computeNotProhibitedSet()`

Fig. 2. Determining operations that are no longer prohibited

`computeNotProhibitedSet(`$(s, r_2), SR_d^+, SR_d^-, \mathsf{C}$`)`. At this point, it is known that the threshold for the constrained set $Q_{c_2}^s$ is no longer attained and that c_2 is no longer violation-prone (for session s). Hence, the activation of role r_3 in s is no longer prohibited, and `computeNotProhibitedSet(.)` returns (s, r_3). The negative authorization policy is updated by deleting this operation that is no longer prohibited (which is returned by `computeNotProhibitedSet(.)`). Specifically, the constraint enforcement relation $SR_d^- \leftarrow SR_d^- \setminus \{(s, r_3)\}$.

D. Evaluation architecture

In order to enforce constraints, it is necessary to evaluate constraints at some point in time. Existing constraint enforcement mechanisms evaluate constraints while deciding whether or not to permit an operation [2], [3]. In other words, such mechanisms decide a constrained operation by performing both authorization checking and constraint checking.

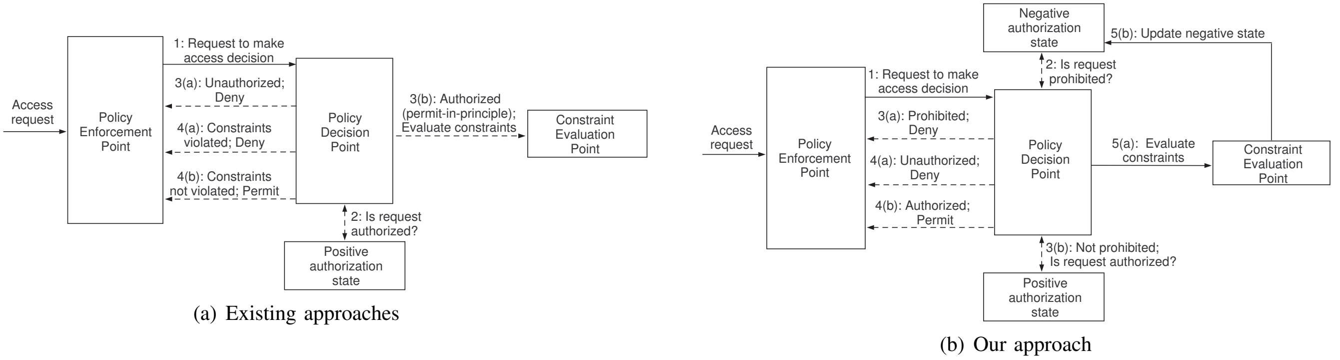
Figure 3 illustrates the system design of existing constraint enforcement approaches and our approach. Figure 3(a) shows the design of the commonly used approaches and Figure 3(b) shows the design of our approach. In these figures, the dotted lines indicate control flow that is part of the decision-making process and the solid lines indicate control flow that is not part of the decision-making process.

Existing constraint enforcement mechanisms (Figure 3(a)) decide a given operation by, firstly, attempting to obtain a *permit-in-principle* authorization from the policy decision point (PDP) by referring to the positive authorization state. An operation is *denied* by the policy enforcement point (PEP) if it is not permitted by the authorization state. Should a permit-in-principle decision be obtained, then constraints are evaluated to check if enforcing a permit-in-principle decision violates any constraint. A request is *denied* if enforcing the permit-in-principle decision violates at least one constraint, and *permitted* otherwise.

In our approach (Figure 3(b)), firstly, the PEP attempts to check whether the operation is prohibited by referring to the negative authorization state. The operation is *denied* if it is prohibited. Otherwise, the PDP attempts to verify whether the operation is authorized by referring to the positive authorization state. The operation is *denied* if it is not authorized, and *permitted* otherwise. Should an operation be permitted, we then evaluate constraints to determine subsequent operations that would either violate or no longer violate any of these constraints. The negative authorization state is updated to reflect the results of constraint checking process.

Existing constraint enforcement mechanisms [2], [3] perform constraint checking as a part of the decision-making process. A fundamental difference of our framework from existing approaches is that constraint checking is *not* part of the decision-making process. Consequently, decision-making times in our framework are lower than existing approaches. Specifically, in order to decide an operation *op*, we simply need to check that *op* is not prohibited and that *op* is sufficiently authorized. It is this feature of our framework that makes access control decisions simpler to compute.

It is important to note that, in order to decide an operation, none of the existing constraint enforcement mechanisms [2], [3] perform constraint checking before authorization checking. This is because performing constraint checking for an unauthorized request would unnecessarily increase decision-making



(a) Existing approaches

(b) Our approach

Fig. 3. A comparison of system designs

TABLE V
A COMPARISON OF ENFORCING A CONSTRAINED RBAC POLICY

Successful operation	**Constraint evaluation**	
	Existing approaches	Our approach
`activateRole`(s, r_1)	✓	✓
`activateRole`(s, r_2)	✓	✓
`activateRole`(s, r_3)	✓	✗

times. This further justifies our approach to altogether exclude constraint checking from the decision-making process.

Furthermore, the number of times constraints are evaluated is reduced in our approach. Consider, for example, a constrained RBAC policy with a constraint $(S, \{r_1, r_2, r_3\}, 2, d)$ and assume that three different operations are to be decided that individually activate roles r_1, r_2 and r_3 in a session s. Existing approaches evaluate the constraint for deciding each operation; whereas, in our approach, the constraint is evaluated only for deciding two operations (because the third operation will be denied by referring to the prohibited authorization state). Table V gives a comparison of existing approaches and our approach for enforcing this policy. More generally, given a constraint $c = (A, B, k, x)$ and a set of n consecutive grant operations (such that $k < n$), which are constrained by c, our approach requires c to be evaluated at most k times; whereas, in existing approaches, c is evaluated n times.

Although, our model simplifies decision-making, the overheads are transferred to the task of maintaining the prohibited authorization policy. Specifically, following the success of a revoke operation, we need to delete tuples from the constraint enforcement relation in order to remove the effect of certain operations, which are no longer prohibited. Such a policy update mechanism is not required in existing models. We may be able to perform batch updates to the prohibited authorization policy at regular intervals, thereby reducing the overall load. However, the prohibited authorization policy might, at certain times, be more stricter than it is required by prohibiting certain operations, which in actual fact must no longer be prohibited. The converse of this may also be true. That is, the prohibited authorization policy might, at certain times, be less stricter than it is required by *not* prohibiting certain operations, which in actual fact must be prohibited.

V. RELATED WORK

The work of Simon and Zurko includes an implementation model for enforcing constraints in role-based environments [3]. In this work, a request is only permitted if the user is authorized to perform the request and no constraint is violated as a result of permitting the request. Essentially, the reference monitor performs both authorization checking and constraint checking for deciding a request.

Chadwick *et al* discussed enforcement of dynamic separation of duty constraints that span across multiple sessions [2]. Such constraints are referred to as historic constraints in our work. In the work of Chadwick *et al*, successful operations are recorded in a repository, which is referred to for enforcing constraints. Specifically, in order to decide an operation, the decision function performs the following two functions. Firstly, the decision function refers to the RBAC authorization structures for determining whether the operation could be granted. Subsequently, the decision function checks whether any constraint prohibits the operation to be granted by referring to the repository of successful operations. Essentially, the former function performs authorization checking and the latter process performs constraint checking.

Crampton proposed an implementation model for enforcing historic separation of duty constraints [6]. This work introduces the concept of a *blacklist*, which is a dynamic access control structure. A blacklist is associated with each user and records negative authorizations of the user. A request initiated by a user u would be denied if the request belongs to the blacklist associated with the user. However, this work only considers enforcement of permission-based historic separation of duty constraints. In comparison, our framework presents a generalized model using a suite of constraint enforcement relations (A^-), which encode prohibited authorization state. Note also that a blacklist is similar to one of our constraint enforcement relations, $UP_h^- \subseteq U \times P$. Hence, we consider blacklists as a special case within our framework.

The Chinese Wall model includes a history matrix $N \subseteq S \times O$, where S is a set of subjects and O is a set of objects [1]. Initially, all entries of the matrix are set to be *false*. A cell

$N[s_i, o_j]$ is set to *true* only if subject s_i has been permitted to access object o_j. In other words, the history matrix records successful access requests. Access decisions are made based on the previous requests that were granted to a subject by referring to the entries of the history matrix. The history matrix of the Chinese wall model and our constraint enforcement relations A^- actually serve a very similar purpose, which is to enforce constraints. Our framework, however, is a dynamic and more complex version of the Chinese wall history matrix.

Our ideas could perhaps be implemented by using XACML policies and triggering policy updates through the obligation mechanism. This mechanism is used to signal to the policy enforcement point that certain actions must be taken, in addition to enforcing the access control decision. In the context of our work these actions would be to update the negative authorization policy. Unfortunately, XACML 2.0 [9] – the latest standard – provides no mechanism for updating XACML policies.

VI. Conclusion

We extended the state of the RBAC96 model to provide support for the evaluation of certain dynamic and historic constraints. We introduced a new constraint syntax that can be used for both separation of duty and cardinality constraints. We then derived various evaluation contexts from the extended RBAC state, which are used to evaluate constraints with different combinations of scope and context. We also defined formal semantics for constraint satisfaction within our model.

We noted that approaches that perform both authorization checking and constraint checking when deciding an operation have performance overheads. The decision functions of large-scale systems, where hundreds of requests arise concurrently, require relatively simple decision-making algorithms. We described a new framework that should reduce access decision times – compared to existing approaches [1]–[3] – for evaluating whether a state transition would result in constraint violations. Essentially, we transformed the constraint checking problem into an authorization checking problem by updating the authorization state when constraints become violation-prone.

We introduced two different relations for representing authorization state: an authorization relation and a constraint enforcement relation. In our framework, a constrained operation is decided by only referring these two relations. We also presented algorithms for updating these authorization relations following the success of both grant and revoke operations.

In future work we would like to develop a test-bed that would enable us to compare the performance of different approaches to constraint enforcement. We would also like to investigate the extent to which it is appropriate to adopt a "lazy" approach to policy updates. Such an approach obviously leads to the possibility of false positives, where a request violates a constraint but is granted because the negative policy has not had relevant prohibited requests appended to it, and false negatives, where a request does violate a constraint but is denied because the negative authorization policy has not had relevant requests removed.

References

[1] D. Brewer and M. Nash, "The Chinese wall security policy," in *Proceedings of 10th IEEE Symposium on Security and Privacy*, 1989, pp. 206–214.

[2] D. Chadwick, W. Xu, S. Otenko, R. Laborde, and B. Nasser, "Multi-session separation of duties (MSoD) for RBAC," in *Proceedings of 1st International Workshop on Security Technologies for Next Generation Collaborative Business Applications (SECOBAP'07)*, 2007, pp. 744–753.

[3] R. Simon and M. Zurko, "Separation of duty in role-based environments," in *Proceedings of 10th IEEE Computer Security Foundations Workshop*, 1997, pp. 183–194.

[4] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

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

[6] J. Crampton, "Specifying and enforcing constraints in role-based access control," in *Proceedings of 8th ACM Symposium on Access Control Models and Technologies (SACMAT'03)*, 2003, pp. 43–50.

[7] V. Gligor, S. Gavrila, and D. Ferraiolo, "On the formal definition of separation-of-duty policies and their composition," in *Proceedings of IEEE Symposium on Research in Security and Privacy*, 1998, pp. 172–183.

[8] T. Jaeger and J. Tidswell, "Practical safety in flexible access control models," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 2, pp. 158–190, 2001.

[9] OASIS, "Extensible access control markup language," 2005, document Status- Approved OASIS standard, Available at http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.