Towards Session-aware RBAC Administration and Enforcement with XACML

Min Xu[1], Duminda Wijesekera[1], Xinwen Zhang [2] and Deshan Cooray[1]
[1]Department of Computer Science
George Mason University
Fairfax, VA, USA
{mxu, dwijesek, dcooray}@gmu.edu
[2] Computer Science Lab
Samsung Information Systems America
San Jose, CA, USA
xinwen.z@samsung.com

***Abstract*—An administrative role-based access control (ARBAC) model specifies administrative policies over a role-based access control (RBAC) system, where an administrative permission may change an RBAC policy by updating permissions assigned to roles, or assigning/revoking users to/from roles. Consequently, enforcing ARBAC policies over an active access controller while some users are using protected resources would result in conflicts: a policy may be in effect in the RBAC system while being updated by an ARBAC operation. Towards solving this concurrency problem, we propose a session-aware administrative model for RBAC. We show how the concurrency problem can be resolved by enhancing the eXtensible Access Control Markup Language (XACML) reference implementation. In order to do so, we develop an XACML-ARBAC profile to specify ARBAC policies, and enforce these polices by building an ARBAC enforcement module and a session administrative module. The former synchronizes with the evaluation of access control requests. The latter revokes conflicting user sessions immediately prior to enforcing administrative operations. Experimental studies show reasonable performance characteristics of our initial enhancement to Sun's reference implementation.**

I. Introduction

The fundamental tenant of role-based access control (RBAC) model [18] is that every role is granted a set of permissions necessary and sufficient to perform the job functions of an individual in an organization. Over the years, many administrative role based access control (ARBAC) models have been proposed [17], [8], [6], [15], [14], following the spirit of administrating an RBAC model using another RBAC model. ARBAC models specify the administrators' privileges with so called *administrative* roles that have permissions to configure the components in an RBAC system, including creating/removing roles, changing permissions granted to roles, and assigning/revoking users to/from roles. Independently, the eXtensible Access Control Markup Language (XACML) [2] has become the standard to specify access control policies for Web Services. In order to specify RBAC policies using XACML, an RBAC profile has been defined in XACML [1]. However, to the best of our knowledge, there is no XACML-ARBAC profile to specify ARBAC policies.

During the processing of developing an XACML-ARBAC profile for managing RBAC systems, we have encountered a set of challenges. Firstly, when an administrator exercises any of those access rights granted under an ARBAC policy, it would result in altering the permissions of a user that may be using resources granted under an already enforced RBAC policy. For safety purposes in many applications, the enforcement of an ARBAC policy would entail immediately changing the permissions to use a resource while a user is accessing it. Secondly, an administrative operation usually updates an RBAC policy, which results in read-write conflicts when the access controller is evaluating a user's request based on the updated policy. The underlying reason for these problems lies in the fact that all ARBAC models focus on defining policies to assign different administrative permissions to different administrative roles, while in practice, enforcing these policies affects the runtime state of the RBAC system which may result in unexpected loss of permissions within ongoing sessions and inconsistent policies configurations.

Towards solving these two problems, we propose a *session-aware* administrative model for RBAC. Based on this model we specify concurrency requirements of an ARBAC model and introduce the concept of lock scope for a role, which captures the *affected* roles when the permissions granted to this role are updated due to administrative operations. We then propose an XACML-ARBAC profile in XACML to specify ARBAC policies. Finally we have implemented our solutions by extending Sun's XACML reference implementation [4]. Specifically, we have developed a special administrative policy enforcement point (A-PEP) that competes for read-write locks for RBAC and ARBAC polices along with the evaluation engine of the access controller. We have also developed a session administrator that terminates all user sessions that are affected due to a pending administrative policy change immediately before its enforcement.

The rest of the paper is organized as follows. Section II briefly describes RBAC and ARBAC essentials. Section III introduces our session administrative model for an RBAC system and concurrency control requirements. Section IV presents our XACML-ARBAC profile and the architecture to enforce this profile in XACML. Section V describes our implementation and Section VI presents some performance characteristics. Section VII presents related work and Section VIII concludes this paper.

II. Preliminaries

A. RBAC and ARBAC

We use the notation $RBAC = (U, O, A, R, P, \leq, U2R, R2P)$ for the model of an RBAC system, where the first four entities are the sets of users, objects, actions, and roles, respectively. P is a subset of $O \times A$, representing the set of permissions. The partial ordering $\leq \subseteq R \times R$ is the role hierarchy. $U2R : U \mapsto 2^R$ and $R2P : R \mapsto 2^P$ are relations that are functional in their first coordinate, modeling user-to-role and role-to-permission assignments. That is, $U2R(u, M)$ and $R2P(r, N)$ are true iff user u is allowed to play the set of roles M and role r can execute the permission set N respectively. We use function $assignPerm(u) = \cup_{r \in U2R(u), r \geq r'} R2P(r')$ to return the set of permissions that a given user obtains through his or her assigned roles.

We base our work partially on ARBAC97 [17] and SARBAC [8], which suggest having a set of *administrative roles* (AR) distinct from *user roles*, and permit these administrative roles to create and remove users, roles, assign and revoke users to (user) roles, and grant and revoke permissions to (user and administrative) roles. ARBAC97 has three sub-models referred to as URA97, PRA97, and RRA97, which controls user-to-role assignments (U2R), role-to-permission assignments (R2P), and role-to-role assignment ($\leq$), respectively. An ARBAC model is defined as follows.

Definition 1 (ARBAC): Let $(U, O, A, R, P, \leq, U2R, R2P)$ be an RBAC model. An administrative RBAC model is a tuple $ARBAC = (U, AO, AA, AR, AP, \leq_A, U2AR, AR2AP)$, where

- $AO = U \cup R \cup U2R \cup R2P \cup \leq$ is the set of administrative objects;
- AA is the set of administrative actions given in Table I;
- AR is a set of administrative roles;
- $AP \subseteq (AO \times AA) \cup (AO \times AO \times AA)$ is the set of administrative permissions;
- $\leq_A \subseteq AR \times AR$ is the administrative role hierarchy;
- $U2AR : U \mapsto 2^{AR}$ is the user-to-administrative role assignment;
- $AR2AP : AR \mapsto 2^{AP}$ is the administrative role-permission assignment.

As defined, administrative objects (AO) in ARBAC include the set of users (U), roles (R), user-to-role ($U2R$) and role-to-permission ($R2P$) mapping and the role inheritance relation ($\leq$) in RBAC, and administrative actions (in Table I) create, update, or destroy these objects. For example, the $AssignUser(u, r)$ and $DeassignUser(u, r)$ operation creates and removes entries in the user-to-role mapping $U2R$, respectively. Each execution of an administrative action changes the RBAC system configuration. Due to space limitation, the formal specification for the administrative operations are not covered in this paper. An administrative permission is an application of an administrative action on one or two appropriate administrative objects.

All administrative operations can be classified into "+" operations and "-" operations. A "+" operation adds elements to existing administrative objects such as assigning a user or granting a permission to a role, while a "-" operation deletes elements such as revoking a user or permission from a role. Different administrative operations invoke different session administrative actions in our session-aware administrative model introduced later.

+ Operations	**- Operations**
AddUser(u)	DeleteUser(u)
AddRole(r)	DeleteRole(r)
AssignUser(u,r)	DeassignUser(u,r)
GrantPermission(r,P)	RevokePermission(r,P)
AddEdge(r^c, r^p)	DeleteEdge(r^c, r^p)

TABLE I: Administrative Operations

III. Session Administrative Model

A. RBAC Session Administration

The RBAC96 [18] and NIST RBAC [9] models include the concept of a session, which is used as artifact to model the applications interaction with the access controller. Specifically, a session is a unique context associated with a user, within which that user activates a subset of assigned roles. Consequently, every activated role belongs to one session, and each session belongs to a unique user. Some primitive session management functions are specified in the NIST RBAC model [9]. However, they are not included in existing ARBAC modes [17], [8], [6], [15], [14]. In order to specify appropriate ARBAC policies for an RBAC system, we define a complete administrative model for session management first.

Definition 2 (Session Administrative Model): Let $(U, O, A, R, P, \leq, U2R, R2P)$ be the model of an RBAC system. A session administrative model is a tuple $SAM = (ACTIVE-S, S-ACTION, U2S, S2R, actRole, actPerms)$, where

- $ACTIVE-S$ is the set of all sessions active at a given system state;
- $S - ACTION = \{CreateSession(u, s), DeleteSession(u, s), ActivateRole(u, s, r), DeactivateRole(u, s, r)\}$ is the set of session administrative actions, where $u \in U$, $r \in R$ and $s \in ACTIVE-S$.
- $U2S : U \mapsto 2^{ACTIVE-S}$ is a function mapping a user to a set of active sessions at a system state;
- $S2R : ACTIVE-S \mapsto 2^R$ is a function mapping an active session to a set of activated roles at a system state;
- $U2S \circ S2R(u) \subseteq U2R(u)$ is the constraint that at a system state, all activated roles of a user is a subset of his or her assigned roles, where $U2S \circ S2R(u) = \cup_{s \in U2S(u)} S2R(s)$;
- $activeRoles(u) = \cup_{s \in U2S(u)} S2R(s)$ is a function mapping a user to a set of activated roles in all active sessions at a system state;
- $activePerms(u) = \cup_{s \in U2S(u), r \in S2R(s), r \geq r'} R2P(r')$ is a function mapping a user to a set of activated permissions at a system state.

Each session administrative action changes the system to a new state, for example, by creating/deleting a session for a user, or activating/deactivating a role within a session.

B. Concurrency Control

Similar to an RBAC model, an ARBAC model defines the configuration of the administrative functions of an RBAC system. However, as stated, any configuration change affects the running system state, which may demand session administrative actions according to application specific requirements. The interaction between session administrative actions and system administrative operations (i.e., the ARBAC operations defined in Section II-A) needs to be specified in order to define safety for an ARBAC model. As one of the major contributions of this paper, we identify the following two concurrency requirements between the session administrative model and the system administrative model for an RBAC system.

Revoke activated role or delete active session immediately Suppose an administrative action $aact \in AA$ changes the state of $RBAC$ to a state $RBAC'$. At a give system state t, if $\exists u \in U, p \in P$, $p \in activePerms(u)|_t \wedge p \notin assignPerms(u)|^{RBAC'}$, then

- $\forall s \in U2S(u)$, if $\exists r \in R, p \in R2P(r) \wedge r \in S2R(r)$, $DeleteSession(u, s)|_t$, or
- $\forall s \in U2S(u)$, if $\exists r \in R, p \in R2P(r) \wedge r \in S2R(r)$, $DeactivateRole(u, s, r)|_t$,

where $assignPerms(u)|^{RBAC'}$ is the set of permissions that user u can activate under $RBAC'$, and $DeleteSession(u, s)|_t$ and $DeactivateRole(u, s, r)|_t$ are session administrative actions at system state t. This requirement specifies that, when $aact$ removes one or more activated permissions of a user in a session at a system state, either the active session should be terminated , or all corresponding roles with the permissions should be deactivated. Obviously, only "-" administrative operations cause these actions in a system.

Delay administrative operations At a give system state t, when a permission is used by a user in an active session, any revocation of this permission from the user by an administrative operation is delayed until the role corresponding to the permission is deactivated, or the session is terminated. Formally, when $aact \in AA$ changes an $RBAC$ model to $RBAC'$, if $\exists u \in U, p \in P, s \in U2S(u)|_t$, and $p \in activePerms(u) \wedge p \notin assignPerms(u)|^{RBAC'}$, then $aact|_{t'}$ when $p \notin activePerms(u)|_{t'}$. That is, the administrative operation $aact$ is executed at system state $t' > t$ where the permission is not activated anymore.

Note that these two requirements can be individually or jointly specified in a particular system, e.g., some permissions need to be immediately deactivated in an active session when they are revoked by an administrative action, while other permissions may delay the execution of an administrative operation. For example, the user session is performing some activity that must be completed, such as waiting for a reply from a remote database or rolling back a transaction. Now, if an administrator wants to remove the permission granted to the user which requires terminating the user session. The enforcement of administrative operation is delayed until the user session ends in order to avoid inconsistency.

When an administrative operation modifies a role, we not only need to manage current active sessions, but also manage any created sessions. This is especially necessary in delayed administrative actions. Specifically, when an administrative operation is delayed, although affected permissions or roles are not deactivated immediately , we need to prevent the user from activating them in a new session. To do this, we lock the affected roles. The administrative operation places write locks on the affected roles to prevent the policy decision point (PDP) from "reading" the roles and other administrative operation from "writing" the roles.

Definition 3 (Lock Scope): Let (U, O, A, R, P, $\leq$, $U2R$, $R2P$) be the model of a RBAC system and $r \in R$ be a role. We define the read scope and write scope of r respectively as $rScope(r) = \{r' \in R | r' \leq r\}$ and $wScope(r) = \{r' \in R | r' \geq r\}$.

As stated in Definition 3, the read scope of a role r includes all its junior roles and itself, and the write scope of r includes all its senior roles and itself. This is because, when a session using a role r may lose permissions if any junior role r' loses its permissions, and therefore needs to ensure that if r' is to lose permissions, then r needs to be deactivated. Conversely, if role r is to lose permissions due to an administrative operation, then all roles senior to r, that is the *write scope* of r must not be allowed to be active. For example in Figure 1, the read lock scope for R3 is {R6, R5, R3}. The write lock scope for R3 is {R0, R1, R2, R3}. Note also that the lock scopes of a role could be changed because of an administrative operation.

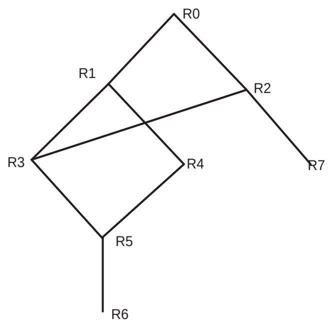

Fig. 1: An example role hierarchy.

We can define the entities affected due to invoking an administrative operation using lock scope. Algorithm 1 in Figure 2 shows how to computer entities affected due to every administrative operation listed in Table I.

IV. XACML-ARBAC Profile and Enforcement Architecture

A. XACML-RBAC Profile

The XACML-RBAC profile 2.0 has been approved as an OASIS standard [1] to specify core and hierarchical compo-

```
Algorithm 1: Compute affected entities
Input: adminOp
Output: Return affected to A-PEP
switch adminOp do
  case DeleteUser(u)
    affected:=u;
  case DeleteRole(r)
    affected:=wScope(r);
  case DeassignUser(u,r)
    affected:=(rScope(r),u);
  case RevokePermission(r,P)
    affected:=wScope(r);
  case DeleteEdge($r^c$, $r^p$)
    affected:=wScope($r^p$);
  otherwise
    affected:=NULL;
return affected;
```

Fig. 2: Compute affected entities of an administrative action.

nents of RBAC models. In this profile, objects, actions, and users are expressed as XACML <Resource>s, <Action>s and <Subject>s. But roles are expressed as <Subject> attributes or <Resource> attributes. This profile also defines three generic XACML policies: a *Permission <PolicySet>*, a *Role* <PolicySet>, and a *Role Assignment <Policy>* or <PolicySet>. These are used to express the remaining entities of an RBAC model (i.e. permissions, $U2R$ and $R2P$ mappings, and a role hierarchy $\leq$), and are briefly explained as follows.

A Permission <PolicySet> is a <PolicySet> used to define a set of permissions associated with a role. It may contain <PolicySetIdReference> to other Permission <PolicySet>s. Stated <PolicySetIdReference>s can be used to inherit permissions of a junior role. Currently, this is the only way to specify the role inheritance in the XACML-RBAC profile.

A Role <PolicySet> binds a set of attributes defining a role in a <Target> to a <PolicySetIdReference> outside of that <Target>. The latter points to the Permission <PolicySet> of the role.

A Role Assignment <Policy> or <PolicySet> does not have a standard specification. The objective of the role assignment <Policy> or <PolicySet> is to specify the user-to-role ($U2R$) assignment. This part of an RBAC policy is supposed to be specified by an entity external to the XACML policy framework, referred to as the *Role Enabling Authority (REA)*. The XACML-RBAC profile does not specify any more requirements for the REA.

B. The XACML-ARBAC Profile

In the OASIS XACML-RBAC profile, roles are defined as attributes of subjects and resources. We enhance the XACML syntax by introducing a new data type *Role*. As our implementation needs to distinguish *administrative roles* from *user roles*, we introduce a *roleType* attribute that can take value from {*userRole*, *adminRole*}. We use all other primitive entities from the XACML-RBAC profile. In particular, the role hierarchy and role-to-permission assignments are expressed in the same way as in the XACML-RBAC profile. We use an XML file to maintain all user-to-role assignments in the policy repository.

We can get all the roles that a user can invoke by querying this XML file. That can be considered a special internal *Role Enabling Authority*. Although we could have maintained the user-to-role assignment as a Role Assignment <PolicySet>, the reason we do not do so is that the current XACML reference implementation does not answer a query such as *What are the roles assigned to Alice?*. Using this extra syntax, we state administrative policies using the same machinery as the RBAC profile, but with the following constraints.

Constraining the Permission <PolicySet> All permissions listed in a <PolicySet> of an administrative role must be administrative permissions. By enforcing the following constraints on the syntax used in a permission <PolicySet>, we ensure that it is an *administrative* Permission <PolicySet>.

1) The <Condition>s are created from applying Boolean operations to existing XACML condition functions and an enlarged set of condition functions listed in Table II.
2) The (<Action>, <Resource>) pair listed in <Rule> must be an AP. That is, the actions must be chosen from operations in Table I.

Constraining the Role <PolicySet> The Role <PolicySet> of an administrative role must be an administrative <PolicySet> with the following additional constraints:

1) All role names that appear in the <Target> of the Role <PolicySet> should be administrative roles.
2) The <PolicySetIdReference> contained in the Role <PolicySet> should point to an administrative Permission <PolicySet>.

C. Enforcing XACML-ARBAC Profile

In order to enforce our XACML-ARBAC profile, we enhance the existing XACML reference implementation with the two entities shown in bold in Figure 3 and explained as follows.

The *Administrative PEP (A-PEP)* receives an administrative access control request, returns a response to the administrator, and if needed, updates relevant polices as a consequence of enforcing the requested administrative operation. The A-PEP functions as a *Role Enabling Authority*. Consequently, when a subject is assigned to a role and revoked from a role, the A-PEP acts as an enabler/disabler by invoking the appropriate administrative operation and updates the U2R mapping in an XML file.

The *Lock Manager* provides the concurrency control necessary to maintain the transactional consistency between simultaneous operations that the PDP requires reading policies in order to evaluate them and the A-PEP needs to modify polices to enforce administrative operations.

D. Concurrency Control

When a non-administrative request arrives at the PDP, the PDP requests a read lock on the policy that is found using the *target matching* algorithm. In case of an administrative request,

Function	Intuitive Meaning
role-exist(r)	check the presence of the role r
inherited-by-assigned-role(r)	check if the given role r is inherited by a role already assigned to the subject
inherit-assigned-role(r)	check if the given role r inherits a role already assigned to the subject
role-assigned-exist(s,r)	check if the subject s is already assigned to the role r
permission-exist(r,p)	check if the role r has been already granted the permission p
role-has-children(r)	check if the given role r has any children
role-has-parent(r)	check if the given role r has any parent
role-is-assigned(r)	check if the given role r is assigned to a user or not
role-is-inherited-by(r1,r2)	check if r1 is inherited by r2
role-is-parent-of(r1,r2)	check if r1 is parent of r2

TABLE II: Extended functions applied in <Condition> in XACML-ARBAC profile

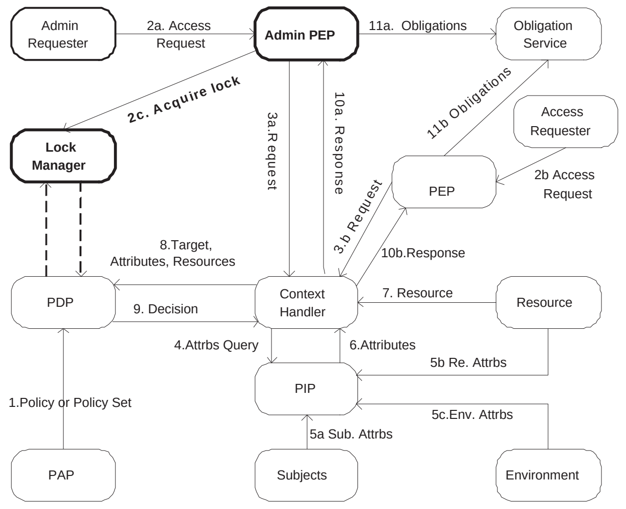

Fig. 3: Extended XACML architecture for XACML-ARBAC enforcement.

the policy evaluation part is similar to the non-administrative request, where the PDP acquires a read lock on the policy for evaluation. If the administrative request is granted, the PDP sends a *request* to the A-PEP. After receiving a *permit* decision from the PDP, the A-PEP acquires a write lock on the policy (recall that administrative requests update XACML policies) that is to be updated. We now describe the details of these steps.

Evaluating Authorization Requests Sun's reference implementation does not alter any XACML policies, and it uses the policy evaluation algorithm explained in [2]. As our enhancements update policies, this evaluation algorithm needs to be protected by a semaphore. Consequently, when a non-administrative request arrives at the PDP, the PDP first requests a read lock (from the *Lock Manager*) on the policy that is found using the *target matching* algorithm, evaluates the request using the existing XACML policy evaluation algorithm, updates the run-time PEP-List (the list of PEPs), and finally releases the read lock on the policy and sends the response back to the requesting PEP, which in turn returns the response back to the user and invokes application dependent activity to enforce the decision. If the PDP fails to acquire the read lock, it returns *indeterminate* as a response to the requesting PEP. The PDP goes through the steps outlined in Figure 4.

```
Algorithm 2: PDP evaluating request
Input: Request, PEPID
Data: PEPList
Output: access control decision
  /*PDP maintains the PEP-List accessible to A-PEP*/
1 policy:=targetMatching(request);
  /*find the policy to be evaluated using target matching*/
2 if AcquireLock(policy,read) then
3    decision:=evaluate(Request,policy);
4    PEPList:=+PEPID;
5    ReleaseLock(policy,read);
6 else
7    decision:=Intermediate;
8 return decision;
```

Fig. 4: PDP evaluation algorithm.

Enforcing Administrative Operations When an administrative request is submitted to the A-PEP, the A-PEP forwards the request to the PDP for evaluation. The PDP uses the same evaluation algorithm used to evaluate the non-administrative request (see Figure 4) and returns the decision to the administrative PEP. If the returned value received at the A-PEP is not a *permit*, the A-PEP conveys the decision to the administrator. Otherwise (e.g., the return value is *permit*), the A-PEP uses the algorithm shown in Figure 5 to enforce that decision. As the algorithm states, if the decision is not a *permit*, the A-PEP returns that decision to the administrator (lines 19). Otherwise, it acquires a write lock on the policy to be updated (line 3), calls the method `getAffected(adminOp)` using the algorithm shown in Figure 2 to determine the parameters that are *affected* by the administrative operation (Line 5). Then, the A-PEP sends a request to all relevant PEPs to terminate user sessions that can be affected by enforcing the enforced administrative operation (lines 6-8), so that updating a policy while these users access permissions granted earlier do not render the access controller unsafe. Because the access controller cannot wait forever for those PEPs to confirm that the requested sessions have been terminated, the A-PEP sets up a timer (line 7). If all those PEPs returned successful answers (lines 12-14), the A-PEP updates the policy to reflect the administrative operation, releases the write lock on the policy (line 16), and finally informs the administrator that the administrative operation is being enforced (the *permit* decision). Conversely, if any PEP fails to return a positive answer when the timer expires, the administrative request is denied.

```
Algorithm 3: Enforcing administrative operations
Input: adminOp, PDPdecision
 /*PDP returns policy decision to A-PEP*/
Data: PEPList
Output: Return decision to administrator
1  if PDPdecision==permit then
2    decision:=deny;
3    if AcquireLock(policy,write) then
4      if adminOp is a (-) operation then
5        Affected:=getAffected(adminOp);
6        forall PEP ∈ PEPList do
7          set(timer, value);
8          sendRequest(PEP,(Affected,killSession));
9        if expires(timer) then
10         acceptFlag:=ok;
11     forall PEP ∈ PEPList do
12       recv(PEP,(Affected, killsSession, NotOK));
13       acceptFlag:=reject;
14     if acceptFlag=ok then
15       modifyPolicy(policy, adminOp);
16       ReleaseLock(policy,write);
17       decision:=permit;
18 else
19   decision:=PDPdecision;
20 return (admin, decision);
```

Fig. 5: Enforcing administrative operations.

E. The Lock Manager

The *Lock Manager* maintains read/write locks on policies, where the PDP is the only potential reader and the A-PEP is the only potential writer of all policies. Because the polices are role-based, the locks are actually placed on the roles. We implemented locking with two atomic operations `AcquireLock(role, read / write)`, `ReleaseLock(role, read / write)` and an `AttemptLock(role, ReadLock, WriteLock)` operation. The method prevents dead-locks and circular locks because all roles are maintained in an ordered list and locks are acquired in the same (increasing) order [12].

V. Prototype Implementation

To show the feasibility and performance of our framework, we have implemented a prototype to enforce the extended XACML profile for ARBAC and concurrency control by augmenting Sun's XACML implementation [4]. Our prototype boots up the access controller with a default administrative XACML policy, which permits the creation of a *super user (SU)* and a *super role (SRole)*, where the *SRole* is the administrative role, assigns SU to SRole, and grants the administrative permissions as shown in Table I to *SRole*. In our implementation, we revoke the conflicting ongoing user sessions immediately prior to enforcing administrative operations.

A. Implementing Condition Functions and Administrative Operations

Sun's reference implementation uses a set of methods, referred to as *condition functions* that compare the retrieved attributes values to expected values to make access decisions. The condition functions provided by Sun's implementation are not capable of checking the conditions for each administrative operation. For example, to add a role r into the system, the access controller needs to check if r is already defined or not. In order to address this deficiencies, we have made two enhancements in our implementation as follows.

In order to check for pre-conditions of each administrative operation, condition functions given in Table II are implemented by extending the function base provided by the existing reference implementation. In each function, we implement the `evaluate` method that is used to evaluate the condition. The input to the condition is provided using `attribute designators` that read information from the request context. In addition, the condition evaluation also requires access to policies, which is provided by initializing each function with a reference to the `policy finder` module of the PDP.

The second is a module used by the A-PEP to modify policies once the PDP permits an administrative operation. This is achieved through a `PolicyManager` that initializes and calls accessor and mutator methods to update the policies. The `AbstractPolicy` class in Sun's reference implementation has been extended with mutator methods as described in Table III. To obtain and update user-to-role assignment, we use standard DOM APIs [5] in order to parse the XML file containing user-to-role assignments.

B. Implementing the Lock Manager

The *Lock Manager* implements a waiting queue with a vector, where index i indicates the i^{th} access request, and serves all requests in the order of submittion. The vector of a waiting process hold semaphores. When a process calls `AcquireLock()`, the semaphore has "memory" if a previous `ReleaseLock()` has been made. Our implementation uses a waiting thread that is woken up when its turn arises in the waiting queue.

VI. Performance Evaluation

The concurrency controller's *waiting queue* implementation slows down the access controller. If the number of administrative operations are executed few and far between, then there is a minimal waiting time for the PDP to request and obtain read locks. However, when an administrative operation is submitted, the total service time becomes the sum of request generation time to the PDP, PDP evaluation time, response building time, lock acquisition time, time to communicate with affected PEPs, time to kill sessions (optional), time to update a policy, and the time taken to release the locks. Thus, when an administrative request is submitted, it delays other user requests that have been submitted after that request. Hence our objective is to evaluate this overall effect on the access controller due to administrative requests.

As a preliminary step towards determining the timing overheads, we build the role hierarchy given in Figure 1. The role hierarchy has eight (8) roles. We grant ten (10) permissions per each of these 8 roles. We assign fifty (50) users per role, and assume that there are ten (10) active user sessions per each role. After building this RBAC policy, the sizes of

Methods	Intuitive Meaning
getInstance(XMLNode)	create an instance of Policy or PolicySet object based on the DOM node
getChild(childId)	return a child of the instance of the Policy or PolicySet
addChild(childId)	add a child to the instance of the Policy or PolicySet
deleteChild(childId)	delete the child from the instance of the Policy or PolicySet
getChildren(XMLNode)	return all children of the Policy or PolicySet
setChildren(XMLNode)	set the child policy tree elements for this node
encode(outputStream)	encode the state of the Policy object to Policy Type XML reprentation

TABLE III: Accessor and mutator methods used in the `PolicyManager`

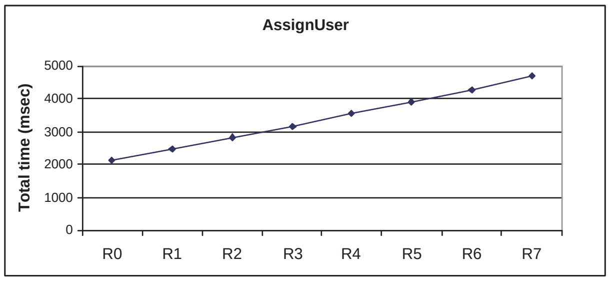

Fig. 6: Latency of `AssignUser`.

our Role <PolicySet>, Permission <PolicySet> and user-to-role assignment file on disk became 12k, 122k, and 41k, respectively.

We simulated the PEP action using method calls where all PEPs take equal time to kill a session. We also placed the PDP, A-PEP, and all other (user) PEPs on the same machine - a 3.4GHz Dual Core Windows XP machine with 1.5G memory. We measured the time taken for administrative operations by calling the Java method *System.nanoTime()* [3]. We executed 8 out of the 10 administrative operations and measured their execution delays, of which we report one in Section VI-A. In addition, we executed one complex operation removing a role from the role hierarchy, which requires executing a series of simple administrative operations. They are described in Section VI-B.

A. Simple Administrative Operations

We built the role hierarchy shown in Figure 1 using our administrative operations. That activity took about 959 msecs to add 8 roles, 844 msecs to add 9 edges, and 711 msecs to grant 10 permissions per each of the 8 roles, and about 3384 msecs to assign 50 users to each role. The average time taken for each simple operation is between 68 to 120 msecs. Out of all these operations, Figure 6 shows the individual time taken for assigning 50 users to each of the 10 roles. We notice that the time grows due to the growth of the U2R mapping. Further analysis shows that this is due to the time taken to parse the XML policy is proportional to the file size.

B. Complex Administrative Operations

The *DeleteRole(r)* assumes that for $r \in ROLE$, no user has activated r in any session and r is not related to any other roles in the hierarchy. Therefore, to remove a role, we need to ensure that these pre-requisites are satisfied by (1) terminating all sessions that have activated r, (2) removing all $(u, r) \in U2R$ for all $u \in USER$, (3) removing all edges (r, r^p) or $(r^c, r) \in \leq$, and then (4) calling the administrative operation *DeleteRole(r)*. Consequently, the time taken to remove a role from the role hierarchy is the sum of time taken to do these individual operation. Accordingly, in order to determine the effect of time taken to delete a role on the number of users permitted to use the role, the number of sessions activating the role and the number of edges connecting the role, we conducted three experiments.

In the first experiment, we fixed the number of users assigned to each role and the number of active sessions of each role. Figure 7 shows the total time taken to delete a role with a fixed number (50) of users permitted to use that role and fixed number of sessions (3) that activated the role given in Figure 1, varying numbers of edges to be deleted. Starting with Figure 1, deleting roles R6 and R7 requires deleting one edge, deleting roles R0 and R4 requires deleting 2 edges, deleting R1, R2, R3, and R5 requires deleting 3 edges. Figure 7 shows that the time taken to delete edges is proportional to the number of edges that need to be deleted.

In the second experiment, we fixed the number of sessions activated by each user at 3, varying numbers of users permitted to activate the role. Figure 8 shows the total amount of time taken to delete each role in Figure 1. Here we assigned 10, 20, 30, 40, 50, 60, 70, and 80 users to R0, R1, R2, R3, R4, R5, R6, and R7, respectively. Figure 8 shows that the total time taken to delete a role is proportional to the number of users that need to be revoked from the role.

In the last experiment, we fixed the number of users assigned to each role at 50, varying numbers of sessions that activate the role. We activated 10, 20, 30, 40, 50, 60, 70, and 80 sessions by R0, R1, R2, R3, R4, R5, R6, and R7, respectively. As Figure 9 shows, the total time taken to remove a role increases with the number of sessions that activate the role.

Several observations stand out form this performance study. First, simple administrative operations execute very fast because they do not affect user activities. Second, the complex operations, especially the *DeleteRole* operation, takes more time because it requires executing a series of administrative operations. For example, in the last experiment, *DeleteRole(R3)* requires executing 50 *DeassignUser* operations, 3 *DeleteEdge* operations, 1 *DeleteRole* operation, and killing 40 sessions. The amortized time for each operation is about 83

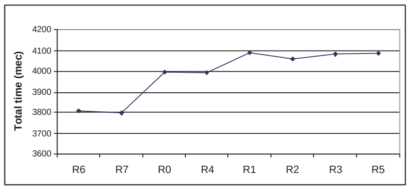

Fig. 7: Effect of # edges on time to remove a role.

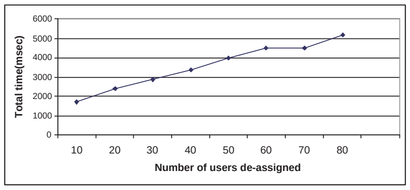

Fig. 8: Effect of # users on time to remove a role.

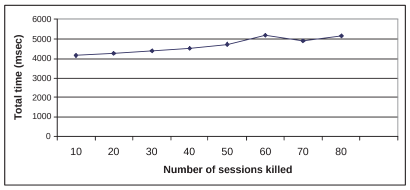

Fig. 9: Effect of # sessions on time to remove a role.

msecs, which is reasonable. Fortunately, deleting a role in a system or organization does not happen often.

VII. Related Work

Over the years, researchers have proposed a plethora of *Administrative Role Based Access Control (ARBAC)* [17], [8], [6], [15], [14] models as the conceptual vehicle to administrate RBAC systems. These models mainly address how to configure U2R, P2R and $\leq$ relationships in an RBAC system. Concurrency control is not addressed in these models.

Seitz et al. [13] present a system permitting controlled policy administration and delegation using the XACML access control system. They use a second access control system *Delegent*, which has delegation capabilities to supervise modifications of the XML-encoded XACML policies. Concurrency issues arising due to administration and policy evaluation is not addressed by them.

Crampton and Chen [7] propose an approach to implement the RBAC model using XACML. They attempt to implement the ANSI RBAC standard [9] using a suit of XACML polices. They use attribute-based role assignment for the U2R assignment, define an XML-based language for specifying separation of duty constraints and propose an extension to the XACML reference architecture in order to enforce these constraints. To the best of our knowledge, these have not been fully implemented.

Concurrency control on XML data has been an active research recently. Haustein et al. [11] introduce a data model called taDOM tree to allow fine-grained locking using a combination of node locks, navigation locks, and logical locks, which we intend to use for our future research.

Janicke et al. [10] propose a concurrent enforcement model for usage control (UCON) [16] policies. Their model separates user, access controller, and system. While their technique enforces concurrency control based on static analysis of dependencies between polices, we resolve concurrency issues during the runtime of a system.

VIII. Conclusion

We propose an enforcement framework for ARBAC policies with XACML. To address concurrency issues, an session-aware administrative model for RBAC is used to manage interactions and potential conflicts between session management and administrative operations. We specify concurrency requirements of an ARBAC model and introduce the concept of lock scope for a role, which captures the affected roles when the permissions granted to this role are updated due to administrative operations. We have developed an XACML-ARBAC profile to specify ARBAC polices and extended Sun's XACML enforcement architecture by introducing an administrative policy enforcement point (A-PEP) and a *Lock Manger* to ensure the safety and integrity of policy management. We have implemented a prototype to enforce the extended XACML-ARBAC profile and demonstrated the feasibility of our framework. Our experimental study shows that our solution encounters a small performance overhead.

References

[1] Core and hierarchical role based access control (RBAC) profile of XACML v2.0, http://docs.oasisopen.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf.

[2] Core specification: extensible access control markup language (XACML), http://www.oasis-open.org/committees/tc_home_php?wg_abbrev=xacml.

[3] Java 2 platform standard edition 5.0, http://java.sun.com/j2se/1.5.0/docs/api/.

[4] Sun's XACML implementation, http://sunxacml.sourceforge.net/.

[5] W3c recommendations, http://www.w3c.org/.

[6] J. Crampton. Understanding and developing role-based administrative models. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[7] J. Crampton and L. Chen. Implementing RBAC and ABRA using XACML. In submission.

[8] J. Crampton and G. Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information and Systems Security*, 6(2):201–231, 2003.

[9] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. Richard Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, August 2001.

[10] H. Janicke, A. Cau, F. Siewe, and H. Zedan. Concurrent enforcement of usage control polices. In *Proceedings IEEE Workshop on Policies for Distributed Systems and Networks (Policy)*, July 2008.

[11] M. Haustein, T. Härder, and K. Luttenberger. Contest of XML lock protocols. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1069–1080. VLDB Endowment, 2006.

[12] H. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, Inc, 1991.

[13] L. Seitz, E. Rissanen, T. Sandholm, B. Sadighi, and O. Mulmo. Policy administration control and delegation using XACML and delegent. In *Grid Computing Workshop 2005*, 2005.

[14] N. Li and Z. Mao. Administration in role based access control. In *ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS)*, March 2007.

[15] S. OH, R. Sandhu, and X. Zhang. An effective role administration model using organization structure. *ACM Transactions on Information and Systems Security*, 9(2):113–137, 2006.

[16] J. Park and R. Sandhu. The UCON$_{\mathrm{abc}}$ usage control model. *ACM Transactions on Information and Systems Security*, 7(1):128–174, February 2004.

[17] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, 1999.

[18] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role based access control models. *IEEE Computer*, 29(2):38–47, 1996.