Towards Session-aware RBAC Administration and Enforcement with XACML

Min Xu
George Mason University
Fairfax, Virginia, USA
mxu@gmu.edu

Duminda Wijesekera
George Mason University
Fairfax, Virginia, USA
dwijesek@gmu.edu

Xinwen Zhang
Samsung Information Systems
America, San Jose, CA, USA
xinwen.z@samsung.com

Deshan Cooray
George Mason University
Fairfax, Virginia, USA
dcooray@gmu.edu

February 4, 2009

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 ongoing user sessions immediately prior to enforcing administrative operations. Experimental shows reasonable performance characteristics of our initial enhancement to Sun's reference implementation.

1 Introduction

One of the fundamental tenants of role-based access control (RBAC) model [22, 13] 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 to achieve this goal [21, 10, 11, 8, 19, 18], 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.

Along our investigation to develop an XACML-ARBAC profile for managing RBAC systems, we have identified a set of challenging issues. 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 is 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 usage of permissions within ongoing sessions and inconsistent policies configurations.

Towards solving these 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 engine [5]. 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 2 briefly describes ARBAC and XACML essentials. Section 3 introduces our session administrative model for an RBAC system and concurrency control requirements. Section 4 presents our XACML-ARBAC profile and the architecture to enforce this profile in XACML. Section 5 describes our implementation and Section 6 presents some performance characteristics. Section 7 presents related work and Section 8 concludes this paper.

2 Preliminaries

2.1 RBAC and ARBAC

We use the notation $RBAC = (USER, OBJECT, ACTION, ROLE, PERMISSION, \leq, U2R, R2P)$ for the model of an RBAC system [1], where the first four entities are the sets of users, objects, actions, and roles, respectively. $PERMISSION$ is a subset of $OBJECT \times ACTION$, representing the set of permissions. The partial ordering $\leq \subseteq ROLE \times ROLE$ is the role hierarchy. $U2R : USER \mapsto 2^{ROLE}$ and $R2P : ROLE \mapsto 2^{PERMISSION}$ 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.

[1] Usually an RBAC model is the configuration of an RBAC system. We do not distinguish them when the context is clear in this paper.

We base our work partially on ARBAC97 [21] and SARBAC [10], 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 as URA97, PRA97, and RRA97, which represent controls over user-role assignment (U2R), permission-role assignment (R2P), and role-role assignment ($\leq$), respectively. An ARBAC model is defined as follows.

definition 1 (ARBAC) *Let* $(USER, OBJECT, ACTION, ROLE, PERMISSION, \leq, U2R, R2P)$ *be an RBAC model. An administrative RBAC model is a tuple* $ARBAC = (USER, A-OBJECT, A-ACTION, A-ROLE, A-PERM, \leq_A, U2AR, AR2AP)$*, where*

- $A - OBJECT = USER \cup ROLE \cup U2R \cup R2P \cup \leq$ *is the set of administrative objects;*
- $A - ACTION$ *is the set of administrative actions given in Table 1;*
- $A - ROLE$ *is a set of administrative roles;*
- $A - PERM \subseteq (A - OBJECT \times A - ACTION) \cup (A - OBJECT \times A - OBJECT \times A - ACTION)$ *is the set of administrative permissions.*
- $\leq_A \subseteq A - ROLE \times A - ROLE$ *is the administrative role hierarchy;*
- $U2AR : USER \mapsto 2^{A-ROLE}$ *is the user-administrative role assignment;*
- $AR2AP : A - ROLE \mapsto 2^{A-PERM}$ *is the administrative role-permission assignment.*

As defined, administrative objects ($A - OBJECT$) in ARBAC include the set of users ($USER$), roles ($ROLE$), user-to-role ($U2R$) and role-to-permission ($R2P$) mapping and the role

+ Operation	**- Operation**
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 1: Administrative Operations

inheritance relation ($\leq$) in RBAC, and administrative actions in Table 1 create, update, or destroy these objects. For example, the $AssignUser$ and $DeassignUser$ operation creates and removes entries in the user-to-role mapping $U2R$, respectively. Each execution of an administrative action changes the RBAC system configuration or condition to a new state. The pre-condition and post-condition of these operations are specified in Appendix A. An administrative permission is an application of 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.

2.2 XACML and Reference Implementation

The eXtensible Access Control Markup Language (XACML) is an XML-based language which specifies access control policies, requests, and responses in distributed computing environments such as Web Services. A request is from a <Subject> (e.g. a user or a process) to perform an <Action> (e.g. read, write) on a <Resource> (e.g. a file or a disk) within an environment (e.g. from a secure machine).

Standard XACML uses three basic elements in constructing access control policies: <Rule>, <Policy>, and <PolicySet>, and allows hierarchical nesting of them. An XACML <Rule> has two elements, a <Condition> and a <Target>, and an *Effect* attribute. The intuitive reading of an XACML rule is that, if the condition of the rule is evaluated to be *true*, the access control decision to perform <Actions> by the <Subjects> on the <Resources> are given by the *Effect* attribute. A <Policy> can consist of a set of <Rule>s. A <PolicySet> holds <Policy>s and other <PolicySet>s. For an access request, the XACML policy evaluation algorithm recursively computes a value from domain {*permit*, *deny*, *nonApplicable*, *indeterminate*} and returns decision in a bottom up manner and uses a collection of (rule and policy) combining algorithms [2] to resolve possible conflicting decisions returned by their sub parts. The OASIS specification [2] identifies four standard combining algorithms: *deny-override*, *permit-override*, *first-one-applicable*, and *only-one-applicable*. For example, the *deny-overrides* algorithm evaluates to *deny* if any applicable rule evaluates to *deny*.

<Target> is a set of simplified conditions for the <Subject>, <Resource>, and <Action> that must be met for a <PolicySet>, <Policy>, or <Rule> to apply to an access request. The <Condition> element further restricts the applicability of the <Rule> already implied by the <Target> in the rule. <Condition>s can be nested using Boolean combinators over other <Condition>s. This can be used to check of the pre-conditions of each administrative operation.

Any <PolicySet> can include one or more <PolicyIdReference> or <PolicySetIdReference> elements. The intended semantics of including a <PolicySetIdReference> in a <PolicySet> is that the content of the referenced <PolicySet> replaces the <PolicySetIdReference> in the referring <PolicySet>. This feature is used in the XACML-RBAC profile [1] to specify role-to-permission assignments and role hierarchies.

Figure 1 shows an example XACML policy that specifies a permission to add a role. This policy has one <Policy> element containing two rules, *Rule "Permission:to:add:a:role"*, and *Rule2*. Line 1 of the policy says that the rule combining algorithm to be used is *permit-overrides*. The policy's target says that this policy is applicable to any subject requesting permission to execute any action on any resource. The target of *Rule 1* narrows the scope of applicable requests to those requesting accesses to the resource *role* with action *AddRole*. The condition of *Rule 1* says that if the role does not exist (computed using our extended function *role-exist*), the request should be permitted. Otherwise, according to *Rule 2* and the rule combining algorithm of the policy, a request applicable to the policy should be denied.

Figure 2 shows the high-level architecture of Sun's XACML reference implementation [5]. In this architecture, the *Policy Administration Point (PAP)* is the entity that creates policies and policy sets; the *Policy Decision Point (PDP)* is the entity that evaluates policies and renders one of {*permit, deny, indeterminate, notApplicable*} as the authorization

```
<Policy PolicyId="add:a:role"
RuleCombiningAlgId="permit-overrides">
  <Target>
    <Subjects><AnySubject/></Subjects>
    <Resources> <AnyResource/></Resources>
    <Actions><AnyAction/></Actions>
  </Target>
  <Rule RuleId="Permission:to:add:a:role" Effect="Permit">
    <Target>
      <Subjects><AnySubject/></Subjects>
      <Resources>
        <Resource>
         <ResourceMatch MatchId="string-equal">
         <AttributeValue DataType="string">role
         </AttributeValue>
         <ResourceAttributeDesignator AttributeId="resource-id"
         DataType="string"/>
          </ResourceMatch>
        </Resource>
      </Resources>
      <Actions>
        <Action>
          <ActionMatch MatchId="string-equal">
            <AttributeValue DataType="string">AddRole
            </AttributeValue>
            <ActionAttributeDesignator AttributeId="action-id"
            DataType="string"/>
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
    <Condition FunctionId="not">
      <Apply FunctionId="role-exist">
        <ResourceAttributeDesignator AttributeId="new-role-id"
         DataType="role"/>
      </Apply>
    </Condition>
  </Rule>
  <Rule RuleID="2" Effect="Deny">
<Policy>
```

Figure 1: An XACML example policy.

decision; the *Policy Enforcement Point (PEP)* is the entity that enforces the access control decision; and the *Context Handler* is the entity that converts native request to one that is in the XACML format (consisting of three components *Subject*, *Resource*, and *Action*) and converts authorization decisions in the XACML format to native formats.

The PAP creates policies at authoring time, e.g., by security administrators. At an access control request time, a subject sends an access request to the PEP as shown in flow 2 of Figure 2. The PEP then forwards this request to the context handler (flow 3) and obtains all the values of the attributes passed in the request. The context handler forms the access control request based on the attributes of the requester, action, resource, and environment, and forwards the request to the PDP (flows 4, 5, 6, 7, 8). PDP uses this information to find the access control policy applicable to the request, which is defined in terms of the attributes of the requestor, action, and resource. The policy can also include functions defined on these attributes. The PDP uses two steps to evaluate the request: it first attempts to find all

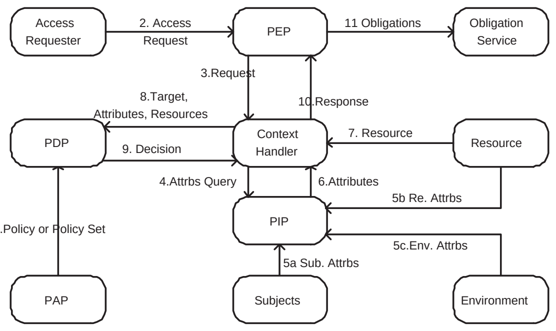

Figure 2: XACML data flow diagram.

the polices applicable to the request by using target matching (flow 1) algorithm, and it then evaluates the rules of the applicable policies and returns its decision back to the PEP via the context handler (flows 9, 10). Finally the PEP enforces the authorization decision.

3 Session Administrative Model

3.1 RBAC Session Administration

The RBAC96 [22] and NIST RBAC [13] models include the concept of session, which is used to interact with applications by users. Specifically, a session is a unique context associated with a user, within which the 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 NIST RBAC model [13]. However, they are not included in existing ARBAC modes [21, 10, 11, 8, 19, 18].

In order to specify appropriate ARBAC policies for an RBAC system, we need to define a complete administrative model for session management first.

definition 2 (Session Administration) *Let ($USER$, $OBJECT$, $ACTION$, $ROLE$, $PERMISSION$, $\leq$, $U2R$, $R2P$) be the model of an RBAC system. A session administrative model is SAM = ($ACTIVE-S$, $S-ACTION$, $U2S$, $S2R$, $actRole$, $actPerms$), where*

- $ACTIVE-S$ *is the set of all active sessions 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 $u \in USER$, $r \in ROLE$ *and* $s \in ACTIVE-S$.

- $U2S : USER \mapsto 2^{ACTIVE-S}$ *is a function mapping a user to a set of active sessions at a system state;*
- $S2R : ACTIVE-S \mapsto 2^{ROLE}$ *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 or equal to the set 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, e.g., by creating/deleting a session for a user, or activating/deactivating a role within a session. The formal semantics of these actions are defined in Appendix B.

3.2 Concurrency Control

Similar to an RBAC model, an ARBAC model defines the configuration of the administrative functions of an RBAC system. However, as aforementioned, 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 2.1) needs to be specified for a safe and complete 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 $aact \in A-ACTION$ changes an $RBAC$ model to $RBAC'$, according to the semantics of Appendix A. At a give system state t, if $\exists u \in USER, p \in PERMISSION$, $p \in activePerms(u)|_t \wedge p \notin assignPerms(u)|^{RBAC'}$, 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 deleted, or all corresponding roles which the permissions are assigned with should be deactivated. Obviously, only "-" administrative operations cause these actions in a system.

Delay administrative operation At a give system state t, when a permission is activated 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 active session is deleted. Formally, when $aact \in A-ACTION$ changes an $RBAC$ model to $RBAC'$, if $\exists u \in USER, p \in PERMISSION, s \in U2S(u)|_t$, 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 some permission 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 suppose that an administrator wants to removed the permission granted to the user which requires to kill the user session. In this case, the enforcement of administrative operation is delayed until the user session ends to avoid inconsistency.

When an administrative operation modifies a role, we not only need to manage current active sessions, but also any new sessions. This is especially necessary in delayed administrative actions. Specifically, when an administrative operation is delayed, although affected permissions or roles are not deactivated at the moment, we need to prevent 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 PDP evaluation "reading" the roles and other administrative operation "writing" the roles.

definition 3 (Lock Scope) *Let ($USER, OBJECT, ACTION, ROLE, PERMISSION, \leq, U2R, R2P$) be the model of a RBAC system and $r \in ROLE$ be a role. We define the read scope and write scope of r respectively as $rScope(r) = \{r' \in ROLE | r' \leq r\}$ and $wScope(r) = \{r' \in ROLE | r' \geq r\}$.*

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 3, the read lock scope for R3 is {R6,R5,R3}. The write lock scope for R3 is {R0,R1,R2,R3}. The lock scopes of a role could be changed because of an administrative operation. For example, the write lock scope for R4 is {R0,R1,R4}. If an administrative role executes the administrative operation *AddEdge(R4,R2)*, the write lock scope for R4 becomes {R0,R1,R2,R4}.

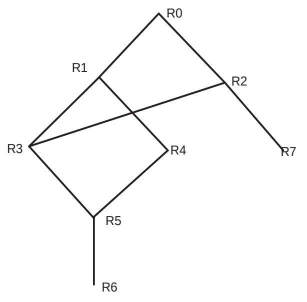

Figure 3: An example role hierarchy.

With the concept of lock scope, we can define the affected entities because of invoking an administrative operation. Algorithm 1 in Figure 4 shows this information for each administrative operation. For example, deleting the role R3 from the role hierarchy in Figure 3 affects all the sessions where R0, R1, R2 and/or R3 are activated. The affected entities are those in *wScope(R3)*.

```
Algorithm 1: Compute affected entities
Input: adminOp
Output: Return affected to A-PEP
1  switch adminOp do
2    case DeleteUser(u)
3      affected:=u;
4    case DeleteRole(r)
5      affected:=wScope(r);
6    case DeassignUser(u,r)
7      affected:=(rScope(r),u);
8    case RevokePermission(r,P)
9      affected:=wScope(r);
10   case DeleteEdge($r^c$, $r^p$)
11     affected:=wScope($r^p$);
12   otherwise
13     affected:=NULL;
14 return affected;
```

Figure 4: Compute affected entities of an administrative action.

4 XACML-ARBAC Profile and Enforcement Architecture

In this section, we present an XACML profile for ARBAC and the architecture to enforce this profile. Because ARBAC is an RBAC model with administrative roles having specialized permissions to administrate an underlying RBAC system, our XACML-ARBAC profile is also an XACML-RBAC profile. We first describe the XACML-RBAC profile and then present our extensions for ARBAC. We then show how the XACML-ARBAC compliant policies can be used to *administrate* the XACML-RBAC policies by executing administrative operations. Finally, we present the architecture to enforce the XACML-ARBAC profile.

4.1 XACML-RBAC Profile

The XACML-RBAC profile 2.0 has been approved as an OASIS standard [1] to specify core and hierarchical components 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 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 of the REA.

4.2 The XACML-ARBAC Profile

In 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 as the following shows:

```
<Subjects>
  <Subject SubjectId="Alice">
    <Roles> <Role>SSO </Role></Roles>
  </Subject>
  <Subject SubjectId="Bob">
    <Roles> <Role>ManagerABC</Role></Roles>
  </Subject>
</Subjects>
```

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 2.

2. The (<Action>, <Resource>) pair listed in <Rule> must be an $A - PERM$. That is, the actions must be chosen from operations in Table 1.

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>.

Sun's reference implementation uses a set of methods, referred as *condition functions*, to compare retrieved attributes values with expected values in order to make access decisions. An example condition function provided by the reference implementation is the form [type]-one-and-only, that accepts a bag of values of the specified type and returns the single value of there is exactly one item in the bag, or an error if there are zero or multiple values in the bag. The condition functions provided by Sun's implementation [5] are not capable of checking the conditions for most administrative operations. For example, to add a role r into the system, the access controller needs to check if r is already defined. Consequently, we add a new set of condition functions listed in Table 2 to support all possible conditional checks for administrative operations.

4.3 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 5 and explained as follows.

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 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 has any children
role-has-parent(r)	check if the given role has any parent
role-is-assigned(r)	check if the give role is assigned 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 2: Extended functions applied in <Condition> in XACML-ARBAC profile

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. Consequently, when needed by the PDP or the context handler, A-PEP provides appropriate instances of the U2R mapping.

The *Lock Manager* provides the concurrency control used 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.

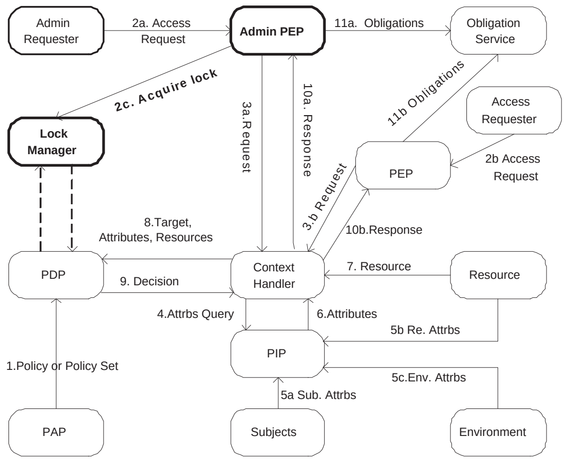

Figure 5: Extended XACML architecture for XACML-ARBAC enforcement.

4.4 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, 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 request updates 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 Section 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, then it evaluates the request uses 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 tho 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 6.

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 as the non-administrative request (see Figure 6) and returns the decision to the administrative PEP. If the returned value received at

```
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;
```

Figure 6: PDP evaluation algorithm.

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 goes through the steps outlined in Figure 7.

Enforcing Administrative Operations When the PDP returns back an authorization decision to the A-PEP, the A-PEP uses the algorithm shown in Figure 7 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 algorithm shown in Figure 4 to determine the parameters that are *Affected* by administrative operation to be enforced (Line 5). Then, the A-PEP sends a request to all PEPs to kill user sessions that can be affected by enforcing the administrative operation (lines 6-8), so that updating a policy while these users access earlier given permission does not render the access controller unsafe. Because the access controller cannot wait forever for those PEPs to confirm that the requested sessions have been killed, the A-PEP sets up a timer to consider all returned values from those PEPs (line 7). If all those PEPs returns 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 enforced (the *permit* decision). Conversely, if all PEPs fail to return a positive answer when the timer expires, the administrative request is denied.

4.5 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. Since 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 that we maintain are in an ordered list and locks are acquired in the same(increasing) order [17].

```
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);
```

Figure 7: Enforcing administrative operations.

5 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 [5]. In this prototype, we revoke the conflicting ongoing user sessions immediately prior to enforcing administrative operations.

5.1 The Birth Process

In our new design, when becomes alive, the access controller follows the *initialization sequence* of creating a *super user (SU)* and a *super role (SRole)*, where the *SRole* is the administrative role. Our prototype boots up the access controller with a default administrative XACML policy, which permits the creation of SU and SRole, assigns SU to SRole, and grants the administrative permissions as shown in Table 1 to SRole.

After the initialization phase, the super user SU is endowed with $SRole$'s administrative permissions described in Table 1. Also as specified, the $SRole$ does not have permissions to delete SU, nor revoke SU from $SRole$. Consequently, permissions granted to $SRole$ remain un-alterable and $SRole$ has no relation with other roles through the role hierarchy, as formally specified in the *AddEdge* administrative operation in the Appendix A.

The access controller does not entertain any user requests during this initialization phase. After the RBAC system boots up, the $SRole$ can create the user roles, create users and assign users to roles, etc. Here we simplify the administrative RBAC system with only a single administrative role.

5.2 Implementing Condition Functions and Administrative Operations

As aforementioned, the condition functions in Sun's reference implementation are not sufficient for enforcing ARBAC profiles. Two enhancements have been made in our implementation. In order to check for pre-conditions of each administrative operation, condition functions given in Table 2 are implemented by extending the function base provided in 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 through `attribute designators` that read information from 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 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 3. To obtain and update user-to-role assignment, we use standard DOM APIs [6] to parse the XML file containing user-to-role assignments.

5.3 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 submitted requests. 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 waiting thread that is woken up when its turn arises in the waiting queue.

6 Performance Evaluation

The concurrency control and its *waiting queue* implementation slow down the access controller. If administrative operations are executed few and far between, 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 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 3. As seen from Figure 3, our 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 our Role <PolicySet>, Permission <PolicySet> and user-to-role assignment file on disk became 12k, 122k, and 41k, respectively.

Our current implementation does not have an elaborate PEP (although we have an A-PEP). Therefore we simulate the PEP action using method calls where the PEP takes an equal time to kill a session. We also place 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 measure the elapse time of administrative operations by calling the Java method *System.nanoTime()* [3]. Under the given conditions, we have experimented with executing the administrative operations. We executed 8 out of the 10 administrative operations and measured their execution delays, of which we report one in Section 6.1. In addition, we executed two other operations of removing some permissions from a role and removing a role from the role hierarchy, which requires executing a series of administrative operations. They are described in Section 6.2.

Methods	Intuitive Meaning
getInstance(XMLNode)	create a 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 3: Accessor and mutator methods used in the `PolicyManager`

6.1 Simple Administrative Operations

We built the role hierarchy shown in Figure 3 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 8 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. A further analysis shows that this is due to the fact that time taken to parse the XML policy is proportional to the file size.

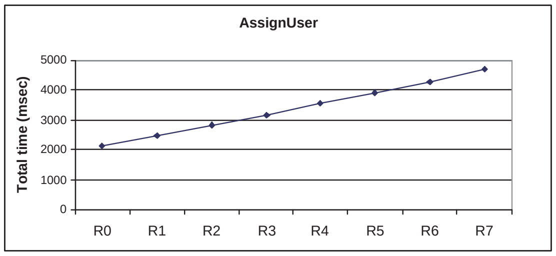

Figure 8: Total time taken to execute `AssignUser`.

6.2 Complex Administrative Operations

We further show the performance characteristics of removing some permissions from a role that has been activated by some users. We also study the performance by removing a role from the role hierarchy while some subjects actively use that role, which invokes a series of administrative operations.

Recall that our definition of *RevokePermission(r,(a,o))* removes the permission *(a,o)* from the role r, provided that no user actively uses r. Consequently, removing any permission, say *(a,o)* must be preceded by killing all sessions that have activated any role in *wScope(r)*, locking all roles in *wScope(r)* so that no other session activates any of them, and then finally revoking the permissions using the administrative operation *RevokePermission(r,(a,o))*.

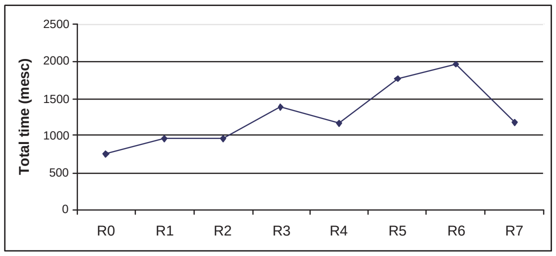

Figure 9: Total time taken to execute `RevokePermission`.

As Figure 9 shows, the time to remove a permission is proportional to the number of sessions that need to be killed in order to lock all roles in *wScope(r)*. For example, revoking a permission from R1 requiring killing 20 sessions, which takes a total of 955 msecs. Revoking a permission from R5 requiring killing 60 sessions, which consumes 1775 msecs. Our observation is that revoking a permission from a role at the bottom of the hierarchy takes more time than at the top of the hierarchy.

Recall that our definition of the *DeleteRole(r)* assumes that for $r \in ROLE$, no user has activated r in any session and the 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 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, Figure 10 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 from the role hierarchy given in Figure 3, with various number of edges to be deleted. Starting with Figure 3, 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 10 shows that the time taken to delete edges is proportional to the number of edges that need to be deleted.

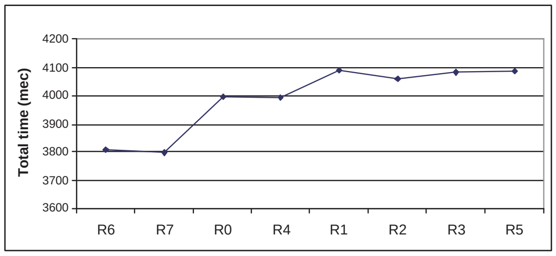

Figure 10: Effect of # edges on time to remove a role.

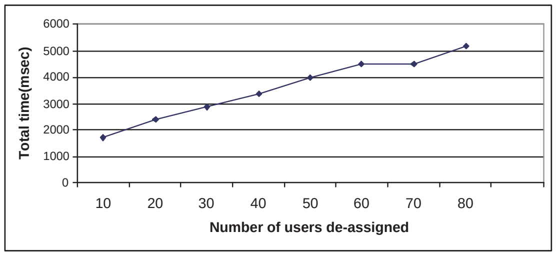

Figure 11: Effect of # users on time to remove a role.

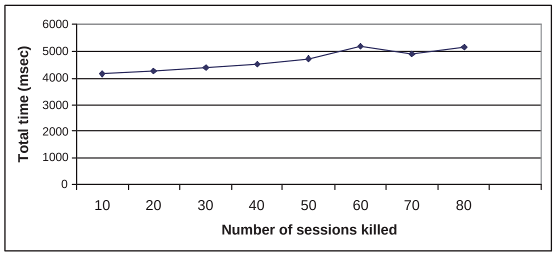

Figure 12: Effect of # sessions on time to remove a role.

In the second experiment, we fixed the number of sessions activated by each user at 3, with various number of users permitted to activate the role. Figure 11 shows the total amount of time taken to delete each role in Figure 3. 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 11 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, with various number of sessions where the role is activated. 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 12 shows, the total time taken to remove a role increases with the number of sessions where the role is activated.

From this performance study, several observations stand out. First, simple administrative operation executes very fast because it does not affect users' activities. Second, the complex operation, especially *DeleteRole* operation, takes quite a bit 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 msecs which is reasonable. Fortunately, deleting a role in a system or organization does not happen often.

7 Related Work

UARBAC [18] proposes a principled approach in designing and analyzing administrative models for RBAC motivated by scalability, flexibility, psychological acceptability, and economy of mechanisms. UARBAC consists of a basic model and one extension: $UARBAC^P$. The basic model adopts the approach of administrating RBAC with RBAC. $UARBAC^P$ adds parameterized objects and constraint-based administrative domains. To the best of our knowledge, UARBAC has not been implemented.

SARBAC [10, 11] extends RBAC administration by adding the concept of *administration scope*s. Administrative scope is defined using the role hierarchy, and is used for defining administrative domains. The administrative scope of a role *(*r) consists of all roles that are descendants of r and are not descendants of any role that is incomparable with r. This definition of scopes works best when the role hierarchy is a tree with an all-powerful root role. In this case, each role's administrative scope is the subtree rooted at that role. When an operation may affect existing administrative domains, ARBAC97 forbids these operations, while SARBAC allows them and handles them by changing existing administrative domains.

One feature of SARBAC is that one simple operation may affect administrative domains of many roles. To the best of our knowledge, SARBAC has not been implemented yet.

NIST [7, 12, 14] has implemented RBAC with an *Administrative Tool* and an RBAC database to store instances of U2R, P2R, and $\leq$ relationships. The administrative tool determines if an update to the three relations stored in the database is permitted by checking the consistency rules, and if so, updates the relationships in the database. This implementation is built for Intranet web servers which is not suitable for distributed applications such as Web Services on the Internet.

Crampton and Chen [9] propose an approach implementing the RBAC model using XACML. They attempt to implement the ANSI RBAC standard [13] 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.

Recently, OASIS XACML v3.0 Administration standard has been approved as an OASIS committee working draft [4]. It describes a profile to express administrative meta-policies which can control different types of polices that individuals can create and modify, but does not use role-based administrative model to manage these XACML policies.

Concurrency control on XML data has been an active research recently. Haustein et al. [16] 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. [15] propose a concurrent enforcement model for usage control (UCON) [20] 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.

8 Conclusion and Future Work

An enforcement framework is proposed in this paper to enforce ARBAC policies with XACML. To address concurrency issues, an elaborate session-aware administrative model for RBAC is used to manage the interaction and 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 the 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 has small performance overhead and can be used for general policy management systems.

One of our ongoing work is to refine the locking granularity for policies. We are also working towards enhancing the A-PEP functionality and creating a richer interface between the PEPs and A-PEP for session management in distributed environments such as Web Services.

References

[1] Core and hierarchical role based access control (RBAC) profile of XACML v2.0, http://docs.oasis-open.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] OASIS XACML v3.0 administration and delegation profile version 1.0, http://www.oasis-open.org.

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

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

[7] J. Barkley, A. Cincotta, D. Ferraiolo, S. Gavrila, and D.R. Kuhn. Role based access control for the world wide web. In *20th National Information System Security Conference*. NIST/NSA, 1997.

[8] J. Crampton. Understanding and developing role-based administrative models. In *Proceedings of the ACM Conference on Computer*

and Communications Security (CCS), November 2005.

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

[10] J. Crampton and G. Loizou. Administrative scope and role hierarchy operations. In *Proceedings of Seventh ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, June 2002.

[11] 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.

[12] D. F. Ferraiolo, J. Barkley, and D.R. Kuhn. A role based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 1(2):201–231, February 1999.

[13] 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.

[14] S. Gavrila and J. Barkley. Formal specification for role based access control user/role and role/-role relationship management. In *Third ACM Workshop on Role Based Access Control*, 1998.

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

[16] 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.

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

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

[19] 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.

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

[21] 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.

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

[23] J. M. Spivey. *The Z Notation: a reference manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

A Formal Specification of Administrative Operations

This appendix formally specifies the suggested administrative operations in terms of pre-conditions and post-conditions using the Z-notation [23]. A value of a data item before the execution of a command (so called pre-state of a data structure) is denoted by a symbol, and its value after the execution of the operation (i.e. the so called post state) is denoted by the same symbol followed by a *prime* (').

AddUser(u): This command creates an RBAC user u.

Pre-condition: u is not already a member of the USER data set.
Formal Specification: $u \notin User$

Post-condition: The User data set is updated. Initially, u is not assigned to any role.
Formal Specification: $User' = User \cup \{u\} \wedge U2R' = U2R$

DeleteUser(u): This command deletes an existing user u from the USER data set. The command is valid if only if u does not have any associated roles.

Pre-condition: u is already a member of the User data set and no roles are assigned to u.
Formal Specification: $u \in USER \wedge \nexists r \in ROLE, M \subseteq ROLE : U2R(u, M) \wedge r \in U$

Post-condition: The USER data set is updated.
Formal Specification: $USER' = USER \setminus \{u\}$

AddRole(r): This command creates a new role r. The command is valid if and only if r is not already a member of ROLE.

Pre-condition: r is not already a member of ROLE.
Formal Specification: $r \notin ROLE$

Post-condition: The new role is added to the role set ROLE set and $U2R, R2P$ remain unchanged. Furthermore, r cannot be assigned to a user until the permissions have been granted to r.
Formal Specification: $ROLE' = ROLE \cup \{r\} \wedge U2R' = U2R \wedge R2P' = R2P$

DeleteRole(r): This command deletes an existing role r from the ROLE data set.

Pre-condition: The role r is a member of the set ROLE, no user is assigned to r and r is not a part of the role hierarchy.
Formal Specification: $r \in ROLE \wedge \nexists u \in USER, M \subseteq ROLE : U2R(u, M) \wedge r \in M \wedge \nexists r' \in ROLE(r \leq r' \vee r' \leq r)$

Post-condition: r is removed from the ROLE data set.
Formal Specification: $ROLE' = ROLE \setminus \{r\}$.

AssignUser(u,r): This command assigns a user u to a role r.

Pre-condition: The user u is a member of the USER data set. The role r is a member of ROLE data set, and the role r is not authorized for that u and is not a child of another role assigned to u.
Formal Specification: $[u \in USER \wedge r \in ROLE] \wedge \, /\exists M \subseteq ROLES[r \in M : U2R(u, M)] \wedge \nexists r' \in ROLE[r' \geq r \wedge r' \in U \wedge U2R(u, M)]$.

Post-condition: The $U2R$ is updated.
Formal Specification: $[U2R(u, M) \rightarrow U2R' = U2R \setminus (u, M) \cup (u, M \cup \{r\})] \wedge [/\exists M \subseteq ROLES U2R(u, M) \rightarrow U2R'(u, \{r\})]$.

DeassignUser(u,r): This command deletes the assignment of the user u from the role r.

Pre-condition: The user u is a member of the USER data set, the role r is a member of ROLE data set and u is assigned to r.
Formal Specification: $u \in USER \wedge r \in ROLE, \exists M \subseteq ROLE : r \in M \wedge U2R(u, M)$

Post-condition: The $U2R$ is updated.
Formal Specification: $\exists M \subseteq ROLES, U2R(u, M) \rightarrow U2R'(u, M \setminus \{r\})$

GrantPermssion(r,(a,o)): This command grants the permission to perform an action a on an object o to a role r.

Pre-condition: The role r is a member of the ROLE data set.
Formal Specification: $r \in ROLE \wedge a \in ACTION \wedge o \in OBJECT$

Post-condition: The $R2P$ is updated.
Formal Specification: $\exists N \subseteq PERMISSION : R2P(r, N) \rightarrow R2P(r, N \cup \{(a, o)\})$

RevokePermission(r,(a,o)): This command revokes the permission to perform action a on an object o from the set of permissions granted to r.

Pre-condition: The role r is a member of the ROLE data set. The permission(a,o) is assigned to the role r.
Formal Specification: $r \in ROLE \wedge \exists N \subseteq PERMISSION : R2P(r, N\{(a, o)\})$

Post-condition: The $R2P$ is updated.
Formal Specification: $\exists N \subseteq PERMISSION : R2P(r, N) \rightarrow R2P' = R2P \setminus (r, N) \cup \{(r, N \setminus \{(a, o)\})\}$.

AddEdge(r^c, r^p)**:** This command makes the role r^c a child role of r^p.

Pre-condition: r^c and r^p are members of the ROLE data set, not related yet and adding does not create cycles in the inheritance hierarchy. *SRole* is not parent or child of any role.
Formal Specification: $r^c, r^p \in ROLE \wedge r^p \nleq r^c \wedge r^c \nleq r^p \wedge r^p \neq SRole \wedge r^c \neq SRole \wedge [\neg \exists r, s \in ROLES(r^c < r < r^P \wedge r^p < s < r^c)]$.

Post-condition: r^p is the parent of r^c.
Formal Specification: $<' = < \cup \{(r^c, r^p)\}$.

DeleteEdge(r^c, r^p)**:** This command deletes an existing child-parent relationship $r^c < r^p$.

Pre-condition: r^c and r^p are members of the ROLE data set and r^p is a parent of r^c.
Formal Specification: $r^c, r^p \in ROLE \wedge [r^c < r^p]$.

Post-condition: The relationship $r^c < r^p$ is deleted.
Formal Specification:$<'=< \backslash\{(r^c, r^P)\}$.

B Formal Specification of Session Administrative Actions

CreateSession(u, s) , creates a new session s for user u. $U2S'= U2S \setminus \{u \mapsto U2S(u)\} \cup \{u \mapsto (U2S(u) \cup \{s\})\}$.

DeleteSession(u, s) , deletes a given session of user u.
$U2S'=U2S \setminus \{u \mapsto U2S(u)\} \cup \{u \mapsto (U2S(u) \setminus \{s\})\}$.

ActivateRole(u, s, r) , adds role r as an active role in a session s of user u. $S2R'= S2R \setminus \{s \mapsto S2R(s)\} \cup \{s \mapsto (S2R(s) \cup \{r\})\}$.

DeactivateRole(u, s, r) , deletes role r from the active role set of session s of user u.
$S2R'=S2R \setminus \{s \mapsto S2R(s)\} \cup \{s \mapsto (S2R(s) \setminus \{r\})\}$.