Administration in Role-Based Access Control

Ninghui Li Ziqing Mao
Center for Education and Research in Information Assurance and Security
and Department of Computer Science
Purdue University
West Lafayette, IN 47907-2107, USA
{ninghui, zmao}@cs.purdue.edu

ABSTRACT

Administration of large-scale RBAC systems is a challenging open problem. We propose a principled approach in designing and analyzing administrative models for RBAC. We identify six design requirements for administrative models of RBAC. These design requirements are motivated by three principles for designing security mechanisms: (1) flexibility and scalability, (2) psychological acceptability, and (3) economy of mechanism. We then use these requirements to analyze several approaches to RBAC administration, including ARBAC97 [21, 23, 22], SARBAC [4, 5], and the RBAC system in the Oracle DBMS. Based on these requirements and the lessons learned in analyzing existing approaches, we design UARBAC, a new family of administrative models for RBAC that has significant advantages over existing models.

1. INTRODUCTION

Role-based access control (RBAC) has established itself as a solid base for today's security administration needs. However, the administration of large RBAC systems remains a challenging open problem. Large RBAC systems may have hundreds of roles and tens of thousands of users. For example, a case study carried out with Dresdner Bank, a major European bank, resulted in an RBAC system that has about 40,000 users and 1300 roles [26]. In RBAC systems of this size, administration has to be decentralized, since it is impossible for a single, fully-trusted administrator, referred to as System Security Officer (SSO) in this paper, to manage the entire RBAC system. Therefore, *delegation* (or decentralization) is an important part of RBAC administrative models that have been proposed in the literature. With delegation, partially-trusted administrators are given the power to change portions of the RBAC state.

Although RBAC itself is relatively well-studied and well-understood, the understanding of decentralized administration of RBAC is still at an early stage. There is a significant gap between the RBAC administration models developed by researchers, namely the ARBAC family [21, 23, 22, 25, 19] and SARBAC [4, 5], and the requirements that have been developed through practical experiences of deploying RBAC, e.g., [12, 26]. For example, one of the most important issues in RBAC administration is how to specify the *administrative domains*. When the SSO delegates the administration privileges to a partially-trusted administrator, the privileges have to be limited to a portion of the RBAC state, which we call an administrative domain. Any RBAC administration approach must provide a mechanism for defining administrative domains. Both the ARBAC family and SARBAC define administrative domains based on role hierarchies, even though they differ in how to use role hierarchies to define administrative domains.

Kern et al. [12] argued that using role hierarchies as a basis for defining administrative domains is problematic in real-life scenarios. One important reason is that the criteria for defining role hierarchies and administrative domains are often different. Administration domains are mostly defined based on the organizational structure, where roles are often defined based on job functions. As observed by many authors, role hierarchies often do not reflect the organizational structure [15, 16, 12]; two roles that are in the same organizational unit may not be related by the role hierarchy in any way. The work by Kern et al. [12] aim at "reconcile the requirements of the actual users of products with research" and "provide researchers with feedbacks in the form of problems encountered by practitioners which should then be addressed more thoroughly".

This paper is motivated by this gap between existing formal models of RBAC administration and the requirements that come up in real deployment. In this paper, we propose a principled approach to reconcile the differences. We identify six design requirements for administrative models of RBAC. These design requirements are motivated by three principles for designing security mechanisms: (1) flexibility and scalability, (2) psychological acceptability, and (3) economy of mechanism. The latter two are in the eight principles for designing security mechanisms identified by Salzer and Schroeder [20]. Using the design requirements that we have developed, we analyze the ARBAC family [19, 21, 22, 25], SARBAC [4, 5], and the RBAC administration approach implemented in the Oracle Database Management System (DBMS).

Based on these requirements and the lessons learned in analyzing existing approaches, we design UARBAC, a new family of administrative model for RBAC. UARBAC has a basic model and one extension. It is the first formal model that adequately addresses the challenges of administering large RBAC systems.

Our principled approach can be contrasted with the approach in developing existing formal models on RBAC, which can be characterized as example-based. In the example-based approach, design decisions are justified through analyzing a few examples and the impact the design decisions have on the examples. While examples are important, the analysis must be guided by high-level security principles. Without guidance from principles, one can often debate whether a particular impact of a design decision is desirable or not. For example, certain side effects of administrative operations are

ASIACCS'07, March 20-22, 2007, Singapore.

considered to be anomalous in ARBAC97, whereas Crampton [4, 5] argued that these side effects are not anomalous. Such issues cannot be resolved satisfactorily without a set of principles. While not everyone will agree with every principle we have identified in this paper, we believe that it is valuable to put forth a set of principles that the community can discuss and debate.

The rest of this paper is organized as follows. We give backgrounds on roles and role hierarchies in Section 2, and describe three existing administrative models in Section 3. We discuss design principles and requirements in Section 4. Section 5 presents UARBAC. We then discuss related work in Section 6 and conclude in Section 7.

2. ROLES AND ROLE HIERARCHIES

RBAC adds the notion of roles as a level of indirection between users and permissions. Roles are created based on job functions and/or qualifications of users. Permissions (i.e., privileges to access resources) are assigned to roles based on the requirements of job functions and/or the entitlement of qualifications. Users are made members of roles based on their job responsibilities and/or qualifications, thereby gaining permissions assigned to those roles. Roles may be organized into a hierarchy, which defines a partial order among roles. We use $r_1 \preceq r_2$ to denote that r_1 is dominated by r_2, and say that r_1 is more junior to r_2, and r_2 is more senior to r_1. This means r_2 inherits all permissions that are assigned to r_1, and all users who are members of r_2 are also members of r_1.

Several existing approaches to RBAC administration use role hierarchies to specify administration domain. Therefore, role hierarchies play a key role in the study of RBAC administration. Figure 1 shows three kinds of role hierarchies that have appeared in the literature. Figure 1(a) shows an example that has been used extensively in the papers on the ARBAC family and SABAC. In fact, it is the only example hierarchy used in these papers. It is well-structured as a lattice, and it has a senior-most role (Dir) and a junior-most role (E). The role Dir has all permissions assigned to any role in the system. Figure 1(b) shows a role-hierarchy in which all roles form an inverted tree. This example is taken from [6]. Such a structure is referred as "limited role hierarchies" in ANSI standard for RBAC [1, 9]. In such hierarchies, the roles towards the bottom are more generic. The more specialized roles inherit permissions assigned to the generic roles, and may be assigned additional permissions. Figure 1(c) shows a layered role-hierarchy taken from [12]. This example came from a real-world example (according to [12]). *Functional roles* are created by asset managers, who control resources. Asset managers also assign permissions for their resources to these functional roles. *Business roles* are created by role administrators, who determine what the functional roles are needed for each business role. This results in a layered, two-level role hierarchy.

3. EXISTING ADMINISTRATIVE MODELS FOR RBAC

In this section, we give an overview of three existing approaches for administering RBAC: the ARBAC family, the SARBAC, and the administrative model implemented in Oracle. We point out that this section is not a standard related work section. We present existing approaches in sufficient details so that, in next section, we can use these to illustrate the principles of designing an RBAC administration model. We will compare these approaches with ours in Section 5.4.

3.1 ARBAC97

To our knowledge, ARBAC97 [21, 23, 22] is the first attempt to specify a comprehensive administrative model for RBAC. ARBAC97 is based on the RBAC96 models [24]. ARBAC97 assumes that there is a set of administrative roles, $\mathcal{AR}$, which is disjoint from the set of normal roles. Only members of these roles can perform administrative operations.

ARBAC97 consists of three sub-models: URA97 for managing user-role assignment, PRA97 for managing permission-role assignment, and RRA97 for managing role-role assignment. All three sub-models rely on the concept of role ranges, which are used as administrative domains. A *role range* specifies a set of roles, and an open role range is written as (x, y), where x and y are normal roles. By definition, $(x, y) = \{r \mid x \prec r \wedge r \prec y\}$. Ranges in URA97 and PRA97 may be open, closed or half-open.

URA97 introduces two relations: can_assign and can_revoke. Each member of can_assign is of the form $\langle a, \varphi, G\rangle$, where a is an administrative role, φ is a pre-condition (discussed below), and G is either a *role range* or a set of explicitly listed roles. The pre-condition, φ, is a propositional logic formula with roles as atoms, and with $\neg, \vee$, and $\wedge$ as logical connectives. For instance, the meaning of the $\langle a, r_1 \vee (\neg r_2 \wedge r_3), G\rangle$ is that a member of a may assign a user u to a role $r \in G$ provided that either u is a member of r_1, or, u is not a member of r_2, but is a member of r_3. Each tuple in can_revoke is of the form $\langle a, G\rangle$, where a is an administrative role, and G is a role range or a set of roles; it means that a member of a can revoke a user from a role in G. In ARBAC97, administration of permissions is perceived as the dual of the administration of users; thus, PRA97 introduces two relations $can\_assignp$ and $can\_revokep$, which are similar to can_assign and can_revoke.

RRA97 introduces five new relations: $can\_assigna$, $can\_revokea$, $can\_assigng$, $can\_revokeg$, and can_modify. The relation can_modify is used to specify how the role hierarchy may change. Each member of can_modify is a tuple $\langle a, G\rangle$, where a is an administrative role, and G is an *encapsulated* range, which is an open range (x, y) that satisfies the condition that given a role $r_1 \in (x, y)$ and a role $r_2 \notin (x, y)$, we have $r_2 \prec r_1$ if and only if $r_2 \preceq x$, and $r_1 \prec r_2$ if and only if $y \preceq r_2$.[1]

The concept of encapsulation is introduced to avoid the situations that some operations performed inside a role range may have some side effects outside the range. To maintain encapsulation of ranges, ARBAC97 further limits what operations are allowed by the tuples in can_modify. For example, to create a new role, one must specify one parent role and a child role. To create the role r with the role x as its child and y as its parent, the range (x, y) must be a *create range*.[2] Only the SSO can create roles without a parent or a child, or outside a create range. Furthermore, any operation such as deleting a role, inserting an edge, and deleting an edge, should not violate the encapsulation of any authority range (see footnote 2). For example, one cannot add the edge QE1$\prec$PL2, even when a tuple in can_modify has the range (ED,DIR), because this new edge would make the two ranges (E1,PL1) and (E2,PL2) not encapsulated anymore. Similarly, one cannot delete the edge

[1] As Crampton and Loizou [4] point out, the original definition for an encapsulated range in [23] implies that no range can be encapsulated. Therefore, we have adopted the correction to the original definition suggested by Crampton and Loizou [23].

[2] Any range reference in the can_modify relation is called an *authority range*. An *immediate authority range* of a role is the unique smallest authority range to which the role belongs. A range (x, y) is a create range if x and y have the same immediate authority ranges or one of x and y is the end point of the other's immediate authority range.

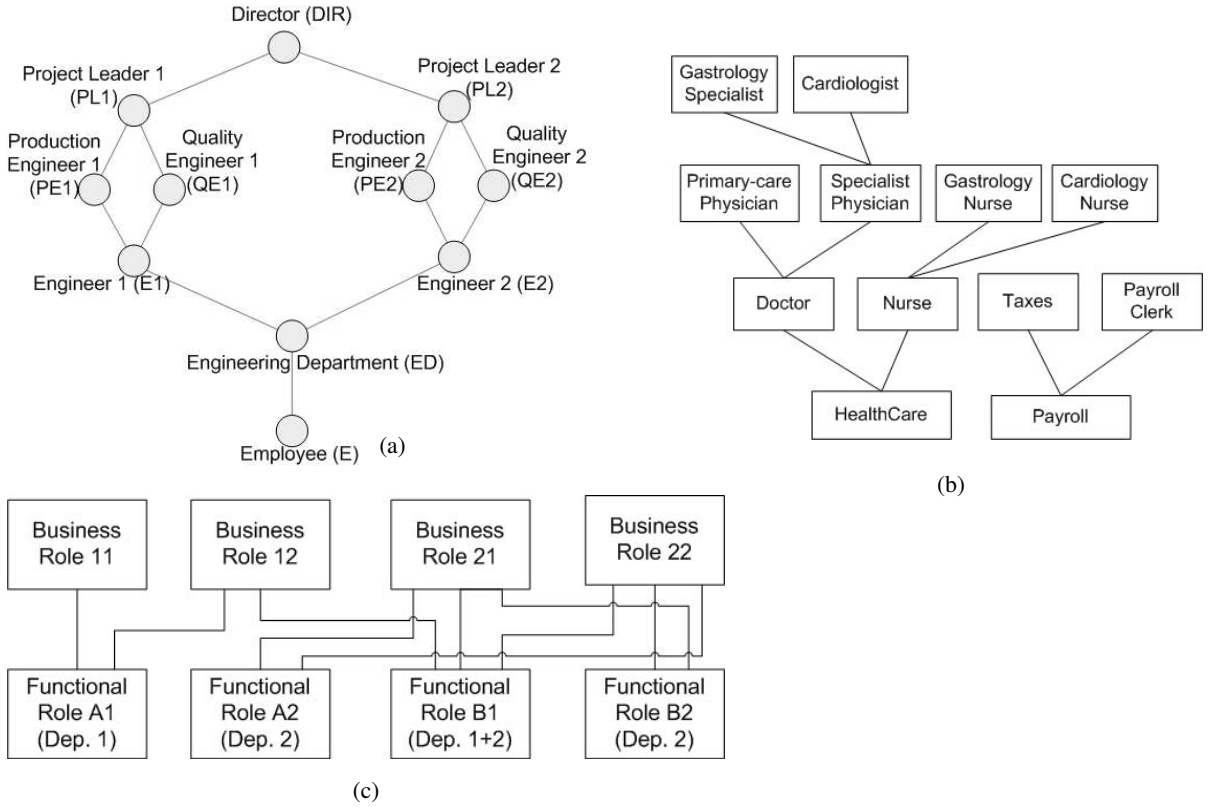


Figure 1: Role hierarchies from RBAC literature.A lattice structured hierarchy is shown in (a). An inverted tree in shown in (b). A layered role hierarchy is shown in (c).

E2$\prec$PE2, because it makes the range (PL2,E2) no longer encapsulated.

We point out some issues of ARBAC97, which will be further discussed in Section 4. First, to administer RBAC, ARBAC97 introduces the notion of administrative roles and nine new relations, which have fairly involved semantics and restrictions. Administrative roles and update to these nine relations are not managed in the model, which implies that they must be managed by the SSO. Second, administrative domains are defined based on role hierarchies. As many operations also change the role hierarchy, some administrative domains may become ill-defined. To avoid this, ARBAC97 introduces concepts such as create ranges and restrictions on what operations can be performed. As a result, a lot of common operations can be performed only by the SSO.

After ARBAC97 was introduced, ARBAC99 [25] and ARBAC02 [19] were introduced to address some perceived shortcomings of ARBAC97. They change only the URA and the PRA components of ARBAC97, but not RRA97. In particular, they do not changed the fundamental approach of using administrative roles, additional relations, and role ranges based on hierarchies.

3.2 SARBAC Family

SARBAC [4, 5] is another important work on RBAC administration; it follows the basic idea of ARBAC family and tries to make the administration more flexible. The administrative model for role hierarchy in SARBAC is later refined and improved by the same author in [3]. We follow the description of SARBAC in [3]. Central to SARBAC is the concept of an administrative scope, which is defined using the role hierarchy, and is used for defining administrative domains. The administrative scope of a role r (denoted by $\sigma(r)$) consists of all roles that are descendants of r and are not descendants of any role that is incomparable with r. More formally, $\sigma(r) = \{s \in \downarrow r : \uparrow s \subseteq \uparrow r \cup \downarrow r\}$, where $\downarrow r$ is the set of roles that are junior to r, $\uparrow s$ is the set of roles that are senior to s. Informally, $r \in \sigma(a)$ if in the role hierarchy every path upwards from r goes through a. Each role is in the scope of the role itself. We say a scope is nontrivial if it includes more than one roles. Using scopes for administration works best when the role hierarchy is a tree, with an all-powerful role at the root; in this case, each role's administrative scope is the subtree rooted at that role. In an inverted role hierarchy such as the one in Figure 1b, no role has a non-trivial administrative scope.

SARBAC uses also administrative roles. To define the administrative domain for each administrative role, SARBAC introduce a relation $admin\_authority$. Each member of $admin\_authority$ is a tuple (a, x), where a is an administrative role, and x is a role. It means the administrative role a has control over the roles in the administrative scope of x. SARBAC allows administration of the $admin\_authority$ relation. This is done by combining the relation $admin\_authority$ and the role hierarchies into an extended hierarchy over both the administrative roles and normal roles. The concept of administrative scope can be extended to the extended role hierarchy. Adding or deleting an $admin\_authority$ relation is equivalent to adding or deleting a corresponding edge in the extended hierarchy. When an operation on the extended hierarchy is done, some updates to $admin\_authority$ happen automatically, in order to preserve the administrative scope and to remove redundancies. Further details of SARBAC are beyond the scope of this paper; readers are referred to [3, 4, 5].

In summary, similar to ARBAC97, SARBAC also uses role hierarchies to define administrative domains. When some operations may affect existing administrative domains, ARBAC97 for-

bids 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.

The administrative model for role hierarchy in SARBAC is later refined and extended to RBAT, a template for role-based administrative models [3]. RBAT formalize the interaction between the role hierarchy operations and the administrative scopes, by having the operations preserve certain aspects of administrative scopes. The role hierarchy administrative model in both ARBAC97 and SARBAC can expressed in terms of the RBAT framework.

3.3 Oracle

The Oracle DBMS implements the notion of roles since early 1990s, and it includes support for administration of the access control state. Unlike ARBAC and SARBAC, Oracle's RBAC administration have been widely used in real world, Oracle thus presents an invaluable reality check for administrative approaches to RBAC. The success of RBAC research is partially due to the fact that the notion of roles has been implemented in commercial systems, so that the research can be guided by real-world experiences. We believe research on administrative models for RBAC must also learn from existing systems such as Oracle.

There are two kinds of privileges in Oracle: *system privileges* and *object privileges*. There are over 100 system privileges in Oracle 10g. For example, the "$create\ role$" system privilege allows one to create a new role, "$drop\ any\ role$" allows to drop any role, "$grant\ any\ role$ allows to grant any role to a user or another role". An object privilege identifies an object, which is either a table or a view, and an access mode, which is one of the following: $select$, $insert$, $update$ and $delete$. Oracle's permission management is a hybrid of DAC (Discretionary Access Control) and RBAC. Privileges can be granted to users and to roles. And roles can be granted to roles and to users. A system privilege or a role can be granted "with admin option". If a user is granted a role with admin option, then we say the user has admin power over the role. This enables the user to grant the role to other users and roles as well as to revoke the role from other users or roles. A role r_1 can also be granted to another role r_2 with admin option, in which case any user that is a member of r_2 has admin power over r_1. A user can create a role if he has the $create\ role$ system privilege and the role to be created does not already exist. When a role is created, the creator will be automatically granted the role with admin option. This enables the creator to further grant the role to any other role or user.

In Oracle, if one has control over a permission, then one can grant the permission to any role; no control over the role is needed. This is different from the approach in ARBAC97 and SARBAC, in which granting a permission to a role is viewed as a dual of assigning a user to a role, and requires the granter has some kind of control over the role. Oracle's design seems more intuitive. Granting a role to a user implies giving out privileges associated with the role; thus some control over the role is needed. Similarly, granting a permission to a role implies giving out the permission; thus some control over the permission (rather than over the role) is needed. On the other hand, Oracle's approach leads to a denial of service attack: Any user who has the "$create\ role$" system privilege can stop other users from logging in. When a user logs in, a set of roles that the user has are activated, as is any role that has been granted to one of these roles. Oracle has a limit on the number of roles that can be activated in a session; if a user has more roles, then the user cannot log in. Oracle has a predefined role called PUBLIC, which is granted to every user and is activated by default. Any user who has the "$create\ role$" system privilege can create a large number of roles and grant them to PUBLIC, resulting in other users unable to log in.

4. DESIGN PRINCIPLES AND REQUIREMENTS

In this section, we present six design requirements for administrative models for RBAC. These requirements are motivated by three principles: scalability and flexibility, psychological acceptability, and economy of mechanism, and they are grouped into three subsections. Several of these requirements came from the drawbacks we have observed in existing approaches to RBAC administrative presented in Section 3. We thus use these requirements to analyze the three approaches and point out these drawbacks.

4.1 Scalability and Flexibility

REQUIREMENT 1. *Support decentralized administration and scale well to large RBAC systems.*

As RBAC's benefits are most pronounced when used in settings with large numbers of users and permissions, we require that administrative models for RBAC are flexible enough to scale to systems of such size. This requires decentralization of operations such as creating users and roles and the ability to define meaningful administrative domains. Each of ARBAC, SARBAC and Oracle supports decentralized administration allows coexistence of multiple administrators having control over portions of the system. However, they all have limitations and do not scale well to large RBAC systems.

ARBAC and SARBAC are designed to work well with particular kinds of role hierarchies, but do not work well with other kinds role hierarchies. As we discussed in Section 2, role hierarchies may take very different forms. For many role hierarchies, there exist very few (sometime zero) nontrivial administrative domains. As a result, a lot of operations cannot be delegated and must be performed by the SSO role. Recall that ARBAC97 requires administration domains to be encapsulated role ranges, and SARBAC uses administrative scopes. In the role hierarchy in Figure 1(b) (a forest of inverted trees), there exists no encapsulated range or non-trivial administrative scope. For example, in Figure 1(c), if we want to create a new role "Head Cardiology" and make it to be more senior to the role "Cardiologist", this operation should performed by the administrator of the Cardiologist Division. But it cannot be achieved in ARBAC and SARBAC. Similarly, in a layered role hierarchy (such as the one in Figure 1(b)), there exist no encapsulated role range or non-trivial administrative scope. A lot of basic administrative operations cannot be delegated for such role hierarchies using ARBAC or SARBAC. For example, if one wants to grant a functional role to a business role, create a new business role, or create a new functional role with multiple business role parents, all these can be done only by the central administrator in ARBAC and SARBAC.

ARBAC97 introduces additional administrative relations that are administered centrally. When applying to large-scale RBAC systems, the size of these relations may be too large to be administered centrally. This limits the scalability of the approach. In Oracle administrative domains are defined by explicitly enumerating objects in the domain. This works fine when the number of roles is limited, which is probably true in most scenarios in which Oracle is used. However, the administration approach in Oracle does not scale to systems that have thousands of roles.

REQUIREMENT 2. *Be policy neutral in defining administrative domains.*

One of RBAC's advantages is policy neutral, so that it can be configured to enforce multiple kinds of policies. An administrative model for RBAC should remain as policy neutral as possible. As the role hierarchy is designed for sharing and aggregation permissions, using the role hierarchy structure to define administrative domains implies a particular kind of policy. For example, in ARBAC97, administrative domains are defined using role ranges, which are roles between two end-points. In SARBAC, administrative domains are defined using scopes, which are roles below a role. In Role Control Center [6], administrative domains are defined using views, which are all roles above a certain role. The existence of these disparate design decisions illustrate that it is a policy decision how (and in fact whether) to use role hierarchy to define administrative domains. Experiences on RBAC deployments reported by Kern et al. [12] also indicate that using role hierarchies is not a natural approach to specifying the domain of an administrator. In particular, domain is often specified based on the structure of an organization, e.g., all roles in one branch of a bank. The roles that belong to one domain may not be related at all in the role hierarchy. Therefore, it is best to decouple administrative domains from role hierarchies. An RBAC administration model should provide a mechanism for defining administrative domains based on other concepts, e.g., organization units.

4.2 Psychological acceptability

This principle means that it is essential that the human interface be designed for ease of use, and, the user's mental image of his protection goals should match the mechanism [20].

REQUIREMENT 3. *Apparently equivalent sequences of operations should have the same effect.*

When two sequences of operations are conceptually equivalent, their effects should be the same. A special case is when one operation can be conceptually viewed as a sequence of more primitive operations, then the effect of that one operation should be equivalent to carrying out the sequence of the primitive operations. Consider an operation that creates a role r with a set P of roles as parents and a set C of roles as children. This operation can be viewed as first creating the role r, then adding each role in P as a parent and each role in C as a child in an arbitrary order. In practice, when one creates a role, one may not be able to determine all the parents and all the children. Thus, one may want to create the role first and then gradually add the relationships. Such equivalence does not hold in ARBAC97 or SARBAC; in particular, one has to specify a parent (and in the case of ARBAC97, a child as well) in creation; otherwise, either the role cannot be created, or will be outside the creator's administrative domain; and further addition of edges cannot be performed. Oracle's design, on the other hand, has this equivalence property.

REQUIREMENT 4. *Support reversibility.*

This implies two requirements. One is that most sequences of operations should be reversible; that is, given a sequence, there should exist a sequence of operations that reverse the effect of the sequence. Having reversibility, if one makes a mistake, one can go back. Certainly, not all operations are reversible; for example, operations such as deleting objects may not be reversible. However, at a minimum, operations that only add things should be reversible. The second requirement is that if an operation has an obvious reversing operation; then the reversing operation should always reverse the effect as expected. For example, if one adds an edge between two roles, and then immediately delete the edge; the system should return to the state before addition.

This is not satisfied by ARBAC97 and SARBAC; they adopt the approach that if a role dominance relationship is removed, then all other role dominance relationships that have been implied are added back. This violates reversibility. Consider the RBAC state in Figure 2(b)(i), which contains the following relationships: $\mathsf{Architect} \succeq \mathsf{Engineer}$. Suppose that when a product is about to be released, one wants engineers to also serve as QAs and adds a temporary relationship $\mathsf{Engineer} \succeq \mathsf{QA}$. This change results in the role hierarchy in Figure 2(b)(ii). After the release, one wants to delete the temporary relationship, expecting the hierarchy to return to the original state in Figure 2(b)(i). After all, the only reason that the Architect role dominates the QA role in Figure 2(b)(ii) is because one wants engineers to be able to serve as QAs and architects are (a kind of) engineers, and now one does not want engineers to be QAs anymore. However, in ARBAC97 and SARBAC, the resulting state would be Figure 2(b)(iii), violating reversibility. We point out that always removing implied role dominance relationships also violates reversibility, as illustrated by Figure 2(a).

Oracle addresses this problem by maintaining all relationships that have been explicitly added. The operation of deleting an edge removes a relationship that has been explicitly added. The actual role hierarchy is inferred from the relationships. This way, one can distinguish an edge that has been explicitly added from one that has been inferred, thereby maintaining reversibility.

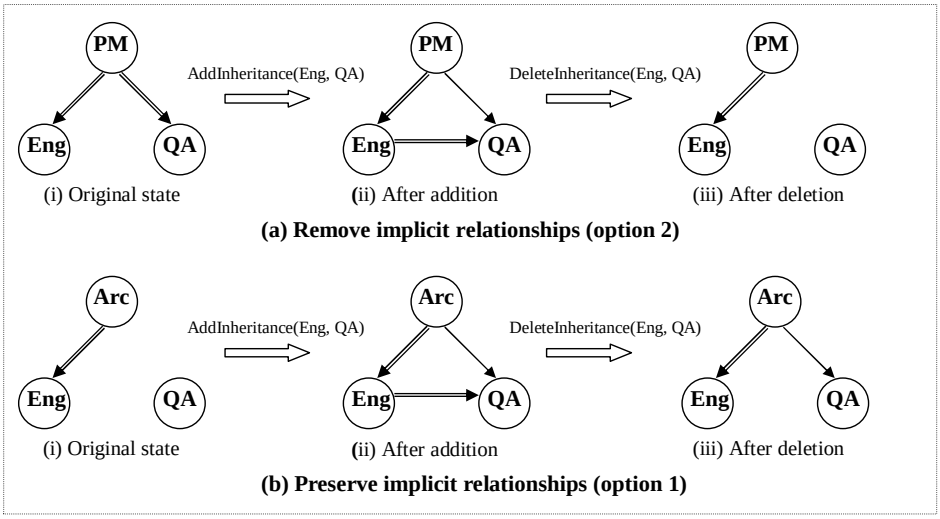
REQUIREMENT 5. *Predictability*

Given a state-change in an administrative model for RBAC, it should be obvious what the effect of that state-change is. That is, there should not be side-effects that are surprising. Otherwise, it is easier for administrators to make mistakes while carrying out administrative operations. In SARBAC, there are automatic updates to the relation $admin\_authority$ following a role hierarchy operation in order to maintain administrative scope and to eliminate redundancy. However, some of "indirect updates" have side-effects that may be considered surprising. For example, in the Figure 1(a), suppose an administrative role PSO1 has control over the administrative scope of Project 1, which is {PL1, PE1, QE1, E1}. When PSO1 delete the role PL1, the original administrative scope for Project 1 breaks up into three trivial administrative scopes {PE1}, {QE1} and {E1}. The system will remove the relationship (PSO1, {PL1, PE1, QE1, E1}) from $admin\_authority$, and make PSO1 have control over the administrative scopes of the intermediate children of the deleted role. It adds two relationships (PSO1, {PE1}) and (PSO1, {QE1}) into $admin\_authority$. Surprisingly, the role E1, which is part of Project 1, becomes outside of the control of PSO1 due to the deletion of PL1, because E1 is not in the administrative scope of either PE1 or QE1. On the other hand, in ARBAC97 and Oracle, the effects of each operation tend to be simpler and easier to understand.

4.3 Economy of Mechanism

REQUIREMENT 6. *Using RBAC to administer RBAC.*

The basic idea of economy mechanism is to keep the design as simple and small as possible. ARBAC97 and SARBAC violate this principle as they all introduce several additional administrative relations with sophisticated semantics. We observe that RBAC systems are used to manage permissions for accessing objects. RBAC systems themselves in turn introduce additional objects, such as users, roles, constraints, the user-role assignment relation, the permission-role assignment relation, and so on. RBAC



ProjManager (PM), Engineer (Eng), Quality Assurance (QA), and Architect (Arc)
$\Rightarrow$ and $\rightarrow$ represent an immediate predecessor relation and a non-immediate partial order relation, respectively.

Figure 2: Adding and deleting a role from RH

administration is about managing operations that change these objects. Based on the principle of economy of mechanism, it makes sense to reuse the mechanisms for managing permissions about accessing objects for managing the administrative operations, rather than introducing additional relations. We call this approach "Using RBAC to administer RBAC".

We point out that ARBAC97 explicitly mentions "using RBAC to administer RBAC" as a design goal. However, the interpretation of "using RBAC to administer RBAC" is different from ours. In ARBAC97, this means that administrative powers are given to administrative roles rather than administrative users directly. However, the relationship between administrative powers and administrative roles are encoded in newly introduced relations. On the other hand, Oracle handles this quite well by using the same mechanism for managing system privileges and roles.

5. UARBAC: A NEW ADMINISTRATIVE MODEL FOR RBAC

We now present our approach for administering RBAC, called the UARBAC family[3] of administrative models for RBAC. UARBAC consists of a basic model and one extension: UARBAC^P, which adds parameterized objects and constraint-based administrative domains. UARBAC adopts the approach of administering RBAC with RBAC. By this, we mean that permissions about users and roles are administered in the same way as permissions about other kinds of objects. An access control system thus has predefined classes of objects for users and roles, as well as other classes of objects that are protected by the access control system. For example, in Security Enhanced Linux [17], 29 classes of objects are defined, including processes, files, etc. Different access modes are applicable for different classes. In this section, we first present a new RBAC model that supports object classes, then describe the basic model in UARBAC and the extension.

5.1 An RBAC model

[3]The letter U in UARBAC does not stand for any thing; or one can consider it to stand for "unnamed".

We now present a new RBAC model that extends the existing RBAC models with the notion of object classes and RBAC schemas.

RBAC Schemas An RBAC schema specifies what kinds of objects are managed by the RBAC system, and what access modes can be applied to these objects. Any RBAC system must be based on an RBAC schema. In our model, an RBAC schema is given by a tuple $\langle C, \mathcal{OBJS}, AM \rangle$.

1. C is a finite set of object classes that the system supports. We require that C contain two predefined object classes: user and role.

2. $\mathcal{OBJS}$ is a function that maps each class in C to a countable set of object names of that class. That is, $\mathcal{OBJS}(c)$ gives all possible names for objects of the class c.

   We use $\mathcal{U}$ to denote $\mathcal{OBJS}(\mathsf{user})$ and $\mathcal{R}$ to denote $\mathcal{OBJS}(\mathsf{role})$, we also require that $\mathcal{R}$ contain a reserved role name sso, for the system security officer role.

3. AM is a function that maps each class $c \in C$ to a set of access modes that can be applied on objects of the class c. There are a few predefined access modes that are relevant to administration.

   Every class has the predefined access mode, admin, i.e., $\forall c \in C, \mathsf{admin} \in AM(c)$. The mode "admin" enables one to delete the object and to give out permissions about the object.

   The access modes for the two predefined classes user and role are fixed by the model as follows.

   - $AM(\mathsf{user}) = \{ \mathsf{empower}, \mathsf{admin} \}$
     The mode "empower" enables one to control who is allowed to add "power" (i.e., permissions) to a user by granting roles to the user. The mode "admin" enables one to delete the user, to give out permissions about the user, and to revoke roles from the user; it also implies the "empower" mode.

- $AM(\mathsf{role}) = \{\ \mathsf{grant},\ \mathsf{empower},\ \mathsf{admin}\ \}$.
  The access mode "grant" over a role enables one to control how this role is granted to other roles and users. The mode "empower" enables one to control who is allowed to add "power" to the role. The mode "admin" enables one to delete the role, to give out permissions (such as grant and empower) about the role, and to revoke roles and permissions assigned to the role; it also implies the modes "grant" and "empower".

There are two kinds of permissions in the RBAC model:

1. *object permissions:* An object permission takes the form $[c, o, a]$, where $c \in C$, $o \in \mathcal{OBJS}(c)$, and $a \in AM(c)$. This permission enables one to access the object o using the access mode a.

2. *class permissions:* A class permission takes the form $[c, a]$, where $c \in C$, and $a \in \{\mathsf{create}\} \cup AM(c)$.

   This permission allows one to access all objects of class c in the access mode a. In particular, $[c, \mathsf{create}]$ allows one to create objects of class c.

   By default, the sso role is granted all class permissions.

How these permissions affect administrative operations will be discussed in Section 5.2.

RBAC States Given an RBAC schema, an RBAC state is given by a tuple $\langle OB, UA, PA, RH\rangle$.

1. OB is a function that maps each class in C to a finite set of object names of that class that currently exist. We have $\forall c \in C$, $OB(c) \subseteq \mathcal{OBJS}(c)$. We use U and R as a shorthand for $OB(\mathsf{user})$ and $OB(\mathsf{role})$, respectively. The set of all permissions, P, is given by

$$P = \{[c, o, a] \mid c \in C \wedge o \in OB(t) \wedge a \in AM(t)\} \cup$$
$$\{\ [c, a] \mid c \in C \wedge a \in AM(c) \cup \{\mathsf{create}\}\ \}\ .$$

2. $UA \subseteq U \times R$ is the user-role assignment relation.

   UA contains the user-role relationships that are explicitly assigned by an administrative operation.

3. $PA \subseteq P \times R$ is the permission-role assignment relation.

   PA contains the permission-role relationships that are explicitly assigned by an administrative operation.

4. $RH \subseteq R \times R$ is an irreflexive and acyclic relation over R. RH contains the role-role relationships that are explicitly added by administrative operations.

   We use $\succeq_{RH}$ to denote the partial order induced by RH, i.e., the transitive and reflexive closure of RH. That $r_1 \succeq_{RH} r_2$ means that every user who is authorized for r_1 is also authorized for r_2 and every permission that is associated with r_2 is also associated with r_1.

We use an example of an RBAC system for files to illustrate our model. In the schema, we have $C = \{\mathsf{file}, \mathsf{user}, \mathsf{role}\}$. The set $\mathcal{OBJS}(\mathsf{file})$ contains all the legal file names; $AM(\mathsf{file}) = \{\mathsf{read}, \mathsf{write}, \mathsf{append}, \mathsf{execute}, \mathsf{admin}\}$. In each state, the set $OB(\mathsf{file})$ contains all the names of the existing files in the system. The object permissions about files takes the form $[\mathsf{file}, o, a]$, where o is an existing file name in $OB(\mathsf{file})$, a is an access mode in AM (file). For example the object permission [file,"/boot.ini", write] enable one to modify the file "/boot.ini", given that "/boot.ini" $\in OB(\mathsf{file})$. The class permissions about files take the form $[\mathsf{file}, a]$, where $a \in AM(\mathsf{file})$. For example, the class permission [file, create] enables one to create a new file; and [file, read] enables one to read any file.

5.2 Administrative operations in UARBAC

The administrative operations in UARBAC are listed in Figure 3. Each administrative operation requires certain permissions to succeed. UARBAC does not fix how to determine the set of permissions that is considered in determining whether an administrative operation is authorized. One way is to use all permissions of the user u who performs the administrative operation, which can be calculated as follows:

$$\mathsf{authorized\_perms}[u] = \{\ p \in P \mid \exists r_1, r_2 \in R$$
$$[(u, r_1) \in UA \wedge (r_1 \succeq_{RH} r_2) \wedge (r_2, p) \in PA]\ \}$$

When the operation is performed in a session, possibly only a subset of all the roles that u is authorized for are activated, one can compute the set of permissions available to the session by considering only the permissions of the roles that are activated in the session.

To grant an object permission $[c, o_1, a]$ to a role r_1, one needs the two permissions $[c, o_1, \mathsf{admin}]$ and $[\mathsf{role}, r_1, \mathsf{empower}]$. This is different from the approach in Oracle, in which, if one controls an object, one can grant permissions about the object to any user or role. To revoke a permission $[c, o_1, a]$ from a role r_1, one needs *either* the permissions $[c, o_1, \mathsf{admin}]$ *or* the permission $[\mathsf{role}, r_1, \mathsf{admin}]$. This design is motivated by the equivalence requirement and the reversibility requirement. Anyone who has the admin permission over an object should be able to delete the object. Before an object is actually removed, permissions about the object need to be revoked from other roles; thus these revocation should be able to succeed. Similarly, anyone who has the admin permission over a role should be able to delete and role, which implies removing all permissions that has been granted to the role. Therefore admin over either the object o_1 or the role r_1 should suffice for revoking the permission $[c, o_1, a]$ from r_1. Under this design, if one is authorized to grant a permission about an object, one is also authorized to revoke the permission, satisfying the reversibility requirement. This design also enables anyone who has the admin permission over a role to control the power of the role, by removing unwanted permissions that have been granted to the role.

Granting/revoking a role to/from a role is similar to granting/revoking a permission to/from a role. To grant a role r_1 to another role r_2 (i.e., making r_2 more senior to r_1), one needs the two permissions $[\mathsf{role}, r_1, \mathsf{grant}]$ and $[\mathsf{role}, r_2, \mathsf{empower}]$. To revoke r_1 from r_2, one needs to satisfy one the following three conditions: (1) has permission $[\mathsf{role}, r_1, \mathsf{admin}]$, (2) has permission $[\mathsf{role}, r_2, \mathsf{admin}]$, and (3) has both $[\mathsf{role}, r_1, \mathsf{grant}]$ and $[\mathsf{role}, r_2, \mathsf{empower}]$. Conditions (1) and (2) are motivated by the need to enable anyone having the admin permission over a role to remove the role. Condition (3) is motivated by the need to enable anyone who can issue a grant to also revoke the grant.

To create new objects of class c, one needs the class permission $[c, \mathsf{create}]$. Unlike in DAC, the creator of an object does not automatically receive "admin" privilege over the object. (Recall that in RBAC permissions cannot be directly assigned to users.) Instead, the creation operation specifies a role r_1 to receive the admin privilege over the object. To do so, the creator should also have the permission $[\mathsf{role}, r_1, \mathsf{empower}]$.

To delete an existing object o_1 of class c, one needs the permission $[c, o_1, \mathsf{admin}]$. When the object o_1 is deleted, all the relation-

Operation	Required Perms	Conditions	Effects
createObject(c, o_1, r_1)	$[c, \mathsf{create}]$, $[\mathsf{role}, r_1, \mathsf{empower}]$	$o_1 \in \mathcal{OBJS}(c)$ $o_1 \notin OB(c)$	$OB'(c) = OB(c) \cup \{o_1\}$, $PA' = PA \cup \{([c, o_1, \mathsf{admin}], r_1)\}$
deleteObject(c, o_1)	$[c, o_1, \mathsf{admin}]$	$o_1 \in OB(c)$	$OB'(c) = OB(c) \setminus \{o_1\}$ Relationships about o_1 are removed.
grantRoleToUser(r_1, u_1)	$[\mathsf{role}, r_1, \mathsf{grant}]$, $[\mathsf{user}, u_1, \mathsf{empower}]$	$r_1 \in R$, $u_1 \in U$	$UA' = UA \cup \{(u_1, r_1)\}$
revokeRoleFromUser(r_1, u_1)	$[\mathsf{role}, r_1, \mathsf{admin}]$ **or** $[\mathsf{user}, u_1, \mathsf{admin}]$ **or** $([\mathsf{role}, r_1, \mathsf{grant}], [\mathsf{user}, u_1, \mathsf{empower}])$	$(u_1, r_1) \in UA$	$UA' = UA \setminus \{(u_1, r_1)\}$
grantRoleToRole(r_1, r_2)	$[\mathsf{role}, r_1, \mathsf{grant}]$, $[\mathsf{role}, r_2, \mathsf{empower}]$	$r_1 \not\geq_{RH} r_2$ $(r_2, r_1) \notin RH$	$RH' = RH \cup \{(r_2, r_1)\}$
revokeRoleFromRole(r_1, r_2)	$[\mathsf{role}, r_1, \mathsf{admin}]$ **or** $[\mathsf{role}, r_2, \mathsf{admin}]$ **or** $([\mathsf{role}, r_1, \mathsf{grant}], [\mathsf{role}, r_2, \mathsf{empower}])$	$(r_2, r_1) \in RH$	$RH' = RH \setminus \{(r_2, r_1)\}$
grantObjPermToRole $([c, o_1, a_1], r_1)$	$[c, o_1, \mathsf{admin}]$, $[\mathsf{role}, r_1, \mathsf{empower}]$	$([c, o_1, a_1], r_1) \notin PA$	$PA' = PA \cup \{([c, o_1, a_1], r)\}$
revokeObjPermFromRole $([c, o_1, a_1], r_1)$	$[c, o_1, \mathsf{admin}]$ **or** $[\mathsf{role}, r, \mathsf{admin}]$	$[c, o_1, a_1] \in PA$	$PA' = PA \setminus \{([c, o_1, a_1], r_1)\}$
grantClassPermToRole(p_1, r_1)	only by the sso role	$(p_1, r_1) \notin PA$	$PA' = PA \cup \{(p_1, r_1)\}$
revokeClassPermFromRole (p_1, r_1)	by the sso role **or** $[\mathsf{role}, r_1, \mathsf{admin}]$	$(p_1, r_1) \in PA$	$PA' = PA \setminus \{(p_1, r_1)\}$

Figure 3: The primitive administrative operations in UARBAC. For each operation, we give the permission(s) the subject that initiates this operation needs to have, the conditions on the RBAC state for the operation succeed, and the effects of each operation by describing which state components are changed. Note that when we say a permission about a specific object is required, the operation will also succeed if the initiator has the corresponding class permission. For example, if $[\mathsf{role}, r_1, \mathsf{grant}]$ is required, the operation will also succeed if the initiator has the class permission $[\mathsf{role}, \mathsf{grant}]$.

ships about o_1 are also removed. Note that if o_1 is a role, then the initiator of the deletion, who has the permission $[\mathsf{role}, o_1, \mathsf{admin}]$, is authorized to revoke all the permissions and roles that are granted to o_1 and all the users and roles that o_1 is granted to. Similarly, if o_1 is a user, then the initiator is authorized to revoke all the roles that are granted to o_1.

The class permissions can be granted only by the sso role. A class permission implies the object permissions over all the objects of that class. For example, the class permission $[\mathsf{role}, \mathsf{grant}]$ enables one to grant any role, i.e., it implies the object permission $[\mathsf{role}, r, \mathsf{grant}]$ for any role r. This is similar to the system privilege "*grant any role*" in Oracle.

We now highlight some of the salient features of our model:

- The RBAC objects, namely users and roles, are treated in the same way as other objects. This is driven by the economy of mechanism principle and the "using RBAC to administer RBAC" requirement.

- The granting operations (namely, grating a permission to a role, granting a role to a role, and granting a role to a user) are all handled in a uniform way, reflecting considerations based on the economy of mechanism principle and the psychological acceptability principle. Granting x to y requires the grant permission over x and the empower permission over y. This is different from many existing approaches, which require only one of the two permissions to perform the operation. Granting x to y naturally requires certain permission over x. UARBAC also requires permission over y for at least two reasons. First, this makes the denial of service attack described in Section 3.3 more difficult to carry out. Even if one can create new roles, one can grant these roles only to those roles over which one holds the empower permission. Second, in a large-scale system, it is natural to restrict the set of users and roles that an administrator can grant permissions to. For example, administrators of one department may be allowed only to grant to users and roles in that department.

  Note that a system in which one needs only grant permission over x to be able to grant x to y (such as Oracle) can be implemented in our model by granting the class permissions $[\mathsf{role}, \mathsf{empower}]$ and $[\mathsf{user}, \mathsf{empower}]$ to all users in the system.

- The grant permission controls both granting and revoking. We made this choice for simplicity, and for supporting reversibility. If a user makes a mistake in granting, the user has the permission to perform the corresponding revocation, cancel the effects of the mistake.

- Here we provide only the most primitive operations. More complex operations can be built from these primitive operations. For example, creating a role with a set of parents and a set of children can be performed by first using createObject to create the role, and then using multiple grantRoleToRole operations.

- In our model, the role hierarchy is not maintained as a partial order among roles. RH is an irreflexive and acyclic relation over roles, which contains only the role-role relationships that explicitly added by an administrative operation. This design provides reversibility to the role hierarchy operations, i.e. grantRoleToRole and revokeRoleFromRole are

mutually reverse. (See the discussions for Requirement 4 in Section 4.2for more explanations.)

- In UARBAC the permissions are assigned only to roles; they are not directly assigned to users. In some situations, one may want to have a hybrid system where permissions can also be assigned to users, (e.g., Oracle). To model such a system, one can extend our model to include a user-permission assignment relation. Granting permissions to users can be administering in the same way was granting permissions to roles.

- In the basic model, administrative domains are defined by explicitly enumerating the objects in the domains. While being flexible, this does not scale well to large RBAC systems; this is addressed in UARBAC^P, which is presented in Section 5.3.

5.3 UARBAC^P: Adding Parameters and Units

Each administrative permission in the basic model is about a single object; in other words, to define an administrative domain, one has to explicitly list all objects in the domain. This does not scale to large RBAC systems. To be able to scale well, we need to be able to define administrative domains based on concepts such as organizational units. The basic idea of the UARBAC^P extension is to assign one or more attributes (called parameters) to each object in the RBAC system, and then define administrative domains using constraints on these parameters. One can view the basic model as a special case in which each object has one parameter, i.e., name, which uniquely identifies the object, and the only kind of constraints is $name = o$, which o denotes an object name.

Schema of Parameterized RBAC: In Parameterized RBAC, an RBAC schema is given by the tuple $\langle D, T, PD, AM\rangle$, which are described below.

1. D is a set of types that are used in the system. Each type defines a set of values, as well as a constraint language for defining sets of values for this type.

   For example, one may define a type for organizational units, which form a tree structure. For instance, in a banking system, the root of the organizational structure is the bank, and children of the root are regions, and each region have some children for branches. A constraint may denote all children under a node, a node plus its children and descendants, and so on. Similarly, one may define a type for files and directories, which are also hierarchical.

2. C is a set of object classes, as in the basic model. However, we allow multiple classes for roles and users. That is, C has two subsets: C_r, which gives the set of all role classes, and C_u, which gives the set of all user classes. This enables us to allow different kinds of roles (e.g., enterprise roles and functional roles) that have different parameters.

3. PD is a function that maps each class in C to a parameter declaration, which takes the form $((pname_1, ptype_1), (pname_2, ptype_2), ..., (pname_k, ptype_k))$. Each pair $(pname_i, ptype_i)$ denotes a parameter, where $pname_1$ is the name and $ptype_i \in D$ is the type of the parameter.

   For example, a role class business_role in C_r may have a parameter declaration $PD(\text{business\_role}) = (name$: ROLE_NAME, $unit$: ORGAN_UNIT), which means that it has two parameters, one is the name of the role, and the other is the organizational unit. Both ROLE_NAME and ORGAN_UNIT are types in D. Suppose that manager is a value in the type ROLE_NAME and 'Branch Hamburg' is a value of type ORGAN_UNIT, then "business_role($name =$ manager, $unit =$ 'Branch Hamburg')" identifies a role.

   Suppose that $PD(c) = ((pname_1, ptype_1), (pname_2, ptype_2), ..., (pname_k, ptype_k))$. An object of class c is identified by $c(pname_1 = s_1, pname_2 = s_2, ..., pname_k = s_k)$, where s_i is a value of type $ptype_i$, for each i in $1..k$. In the following we use $c(\vec{s})$ to represent such an object.

4. AM is a function that maps each class to a set of access modes, and the access modes for a user class or a role class are fixed by the model, as in the basic model.

Using the notion of administrative domains, both object permissions and class permissions become special cases of the following more general parameterized permissions: A *parameterized permission* takes the form $[c, \varphi, a]$, in which c is an object class, φ is a constraint, and $a \in AM(c) \cup \{\text{create}\}$. The constraint φ defines a set of objects of class c, whose parameter values satisfy the constraint. The permission $[c, \varphi, a]$ allows one to access all the objects in the set defined by φ using the access mode a. For example, the parameterized permission [business_role, $unit \leq$ 'Branch Hamburg', create] allows one to create a business role with the parameter $unit$ having a value that is a descendant of 'Branch Hamburg'.

In UARBAC^P, the five operations deleteObject, grantRoleToUser, revokeRoleFromUser, grantRoleToRole and revokeRoleFromRole are similar to those in the basic model. One can treat a parameterized permission $[c, \varphi, a]$ as implying all object permissions $[c, \vec{s}, a]$ where $\vec{s}$ satisfies φ. To create a new object, the creator should specify the parameter values of the object. The operations about *object permissions* and *class permissions* in the basic model are replaced by two operations about *parameterized permissions*. In UARBAC^P, PA is extended with a third parameter, id; that is, when a permission is granted to a role, one also specifies an id value, which needs to be unique among all permissions granted to that role. When one revokes a permission from a role, one specifies the id value of the permission to be revoked. To grant a parameterized permission $[c, \varphi_1, a_1]$ to a role, besides the empower permission over the role, one should have the admin permission over the set of objects defined by φ_1. In other words, one should have the permission $[c, \varphi_2, \text{admin}]$ such that φ_1 logically implies φ_2, which means that every object that satisfies φ_1 also satisfies φ_2 and thus φ_2 describes a larger set. A new operation changeParameters is introduced to change the parameter values of an existing object. In order to do so, the initiator should have the admin permission over the object with the old parameter values and the create permission over the object with the new parameter values. These changed and newly added operations are described in Figure 4.

In summary, in UARBAC^P, administrative domains are defined using constraints on parameter values of objects. This is flexible as well as policy neutral, as it enables one to define administrative domains based on the organizational structure as well as other criteria. Furthermore, parameterized administrative permissions can be further delegated, which further improves scalability.

5.4 Comparisons with existing models

We now review how UARBAC^P meets the requirements stated in Section 4 and then present a comparison of the three approaches described in Section 3 with it.

Operation	Required Permissions	Effects
createObject($c(\vec{s}_1)$, role($\vec{s}_2$))	$[c, \varphi_1, \mathsf{create}] : \vec{s}_1$ satisfies φ_1 $[\mathsf{role}, \varphi_2, \mathsf{empower}] : \vec{s}_2$ satisfies φ_2	$OB'(c) = OB(c) \cup \{c(\vec{s}_1)\}$, $PA' = PA \cup \{([c, \vec{s}_1, \mathsf{admin}], \mathsf{role}(\vec{s}_2))\}$
changeParameters $(c(\vec{s}_1), c(\vec{s}_2))$	$[c, \varphi_1, \mathsf{admin}] : \vec{s}_1$ satisfies φ_1 $[c, \varphi_2, \mathsf{create}] : \vec{s}_2$ satisfies φ_2	Replace all the occurrences of $c(\vec{s}_1)$ in the state with $c(\vec{s}_2)$.
grantPermToRole $([c, \varphi_1, a_1], \mathsf{role}(\vec{s}_1), id_1)$	$[c, \varphi_2, \mathsf{admin}] : \varphi_1$ logically implies φ_2 $[\mathsf{role}, \varphi_3, \mathsf{empower}] : \vec{s}_1$ satisfies φ_3	$PA' = PA \cup \{([c, \varphi_1, a_1), \mathsf{role}(\vec{s}_1), id_1)\}$
revokePermFromRole $(\mathsf{role}(\vec{s}_1), id_1)$	$[c, \varphi_2, \mathsf{admin}] : \varphi_1$ logically implies φ_2, where $([c, \varphi_1, a_1], \mathsf{role}(\vec{s}_1), id_1) \in PA$, **or** $[\mathsf{role}, \varphi_3, \mathsf{admin}] : \vec{s}_1$ satisfies φ_3	$PA' = PA \setminus \{([c, \varphi_1, a_1), \mathsf{role}(\vec{s}_1), id_1)\}$

Figure 4: The changed primitive administrative operations for parameterized extension.

- Requirement 1. Support decentralized administration and scale well to large RBAC systems.

  UARBAC^P supports decentralized administration by allowing multiple administrators to have control over their own administrative domains. As the administrative domains in UARBAC^P are defined based on attributes of objects and are independent of the role hierarchy, it is flexible to support all forms of role hierarchies. UARBAC^P does not introduce any additional administrative relations that require central administration. The parameterized administrative permissions can be further delegated. These features make UARBAC^P more flexible and scalable than existing models.

- Requirement 2. Be policy neutral in defining administrative domains.

  In UARBAC^P, the administrative domains are defined using constraints on parameters of objects. Because the parameter declarations and constraints are defined by administrators, UARBAC^P provides a mechanism for defining administrative domains based on application-level attributes. Therefore, in UARBAC^P how to define administrative domains is a policy decision according to applications.

- Requirement 3. Apparently equivalent sequences of operations should have the same effect.

  Observe that the specification of administrative operations of UARBAC^P includes only the most primitive operations. These primitive operations are carefully designed so that more complex operations can be built from them. This ensures that when one operation can be conceptually viewed as a sequence of more primitive operations, then the effect of that one operation will be equivalent to carrying out the sequence of the primitive operations. For example, deleting an object can be viewed as first removing all relationships that involve the object and then removing the object.

- Requirement 4. Support reversibility

  In UARBAC^P, most operations support reversibility. Each grant operation has a corresponding revoke operation so that one can use the same permissions to go back after making a mistake. The operation createObject has deleteObject as the reversing operation. The reversing operation of changeParameters$(c(\vec{s}_1), c(\vec{s}_2))$ is changeParameters$(c(\vec{s}_2), c(\vec{s}_1))$. The operations that do not have a reversing operation are destructive in nature, e.g., deleting an object, or revoke a permission (or role) from a role (or a user).

- Requirement 5. Predictability

  In the specification of UARBAC^P, the effects of the administrative operations are straightforward and simple. There does not exist any surprising side-effects. For example, the effects of grant (revoke) operations are simply adding (removing) a relationship to (from) the corresponding relations.

- Requirement 6. Using RBAC to administer RBAC

  In UARBAC^P, users and roles are treated in the same way as other objects. And permissions about users and roles are administered in the same way as permissions about other kinds of objects. UARBAC^P does not introduce any additional objects and relations for administration. The administration is unified into the RBAC system.

We summarize the comparisons among the three existing models and the UARBAC family with respect to the six design requirements in Figure 5. Note that Oracle RBAC satisfies four requirements. It uses enumerating objects to define administrative domains, and thus does not scale well to large number of roles. It also imposes two policy decisions in administering RBAC. The first is that anyone can assign a role x to a user or a role, as long as one has admin privilege over x. This leads to potential of DoS attacks and doesn't enforce least privilege in a large enterprise. The second policy decision is that a user must be a member of a role (thus can user the role) before one can administer the role. This decreases the flexibility in enforcing Separation of Duty principles (e.g., one may want to separate the privilege of administering a role from that of using a role). In UARBAC, this can be achieved by not giving an administer user the empower permission over herself. The main innovations of UARBAC are approaches to address the above limitations of Oracle RBAC, including an approach for defining administrative domains that is both flexible and scalable, a way to handle all objects uniformly, and the extra control on granting. Some design decisions we have made in UARBAC can also be viewed as policy decisions. For example, the "grant" permission over a role controls both granting and revoking of the role. (This is the same as the Oracle design.) These decisions are guided by the six design requirements we have identified.

6. OTHER RELATED WORK

The papers that are most closely related to our paper are the ARBAC series [21, 22, 23, 25, 19], the work by Crampton and Loizou [4, 5, 3], and the RBAC system in Oracle. They were discussed in detailed in Sections 3 and 4 and comparison of our approach with them is given in Section 5.4.

In the rest of this section, we briefly discuss other papers in the RBAC literature that are related to our work. The notion of roles was first introduced to access control in the context of database security [27, 2] as a means to group permissions together to ease security administration. The term "Role Based Access Control" was

Requirements	ARBAC, SARBAC	Oracle	UARBAC^P
Decentralized administration & scalability	Partially. Not flexible to support disparate role hierarchies. Introduce central administered relations.	Partially. Not scalable to large number of roles.	Yes. Flexible to support disparate role hierarchies. No centrally administered relations. Administrative domain based on object attributes.
Policy neutral	No. Administrative domains based on the role hierarchy, with fixed policy.	Partially. No control on empowering a user or a role. Must be member of a role to admin a role.	Yes. Define domains based on enumerating objects and application-level attributes.
Equivalence	No	Yes	Yes
Reversibility	No	Yes	Yes
Predictability	ARBAC (Partially): Understandable but complicated. SARBAC (No): Have surprising side-effects.	Yes	Yes
Economy of mechanism	No. Introduce additional relations and objects.	Yes. Same mechanism for managing system privileges and roles.	Yes. Treat users and roles in the same ways as other objects.

Figure 5: Comparisons among the three existing models and UARBAC^P

first coined by Ferraiolo et al. [8, 7]. Sandhu et al. [24] developed the influential RBAC96 family of RBAC models. Nyanchama and Osborn developed the role-graph model [18]. Recently, a standard for RBAC has been proposed and adopted as an ANSI Standard [1, 9]. Parameterized roles have been used before in [10, 13]; however, they have not been used in the context of RBAC administration before.

Using RBAC in enterprise setting and their administration have been studied in [6, 12, 11, 14, 26], these papers report invaluable experiences from deploying large RBAC systems in practice, even though they do not provide formal models for RBAC administration. Our model is largely inspired by these experiences. RBAC administration is also studied in [28, 29, 30]. Our work differs from them in that we adopt a principled based approach and decouple administrative domains from the role hierarchies.

7. CONCLUSIONS

We propose a principled approach in designing and analyzing administrative models for RBAC. We have identified six design requirements for administrative models of RBAC. These design requirements are motivated by three principles for designing security mechanisms: (1) flexibility and scalability, (2) psychological acceptability, and (3) economy of mechanism. We have also used these requirements to analyze several approaches to RBAC administration, including ARBAC97, SARBAC, and the RBAC system in the Oracle DBMS. Based on these requirements and the lessons learned in analyzing existing approaches, we design UARBAC, a new family of administrative model for RBAC that has significant advantages over existing models.

Acknowledgement

This work is supported by NSF CNS-0448204 (CAREER: Access Control Policy Verification Through Security Analysis And Insider Threat Assessment), and by sponsors of CERIAS. We thank Mahesh V. Tripunitara for helpful discussions. We also thank the anonymous reviewers for their helpful comments.

8. REFERENCES

[1] ANSI. American national standard for information technology – role based access control. ANSI INCITS 359-2004, Feb. 2004.

[2] R. W. Baldwin. Naming and grouping privileges to simplify security management in large databases. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 116–132, May 1990.

[3] J. Crampton. Understanding and developing role-based administrative models. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 158–167, Nov. 2005.

[4] 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)*, pages 145–154, June 2002.

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

[6] D. F. Ferraiolo, R. Chandramouli, G.-J. Ahn, and S. Gavrila. The role control center: Features and case studies. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies*, June 2003.

[7] D. F. Ferraiolo, J. A. Cuigini, and D. R. Kuhn. Role-based access control (RBAC): Features and motivations. In *Proceedings of the 11th Annual Computer Security Applications Conference (ACSAC'95)*, Dec. 1995.

[8] D. F. Ferraiolo and D. R. Kuhn. Role-based access control. In *Proceedings of the 15th National Information Systems Security Conference*, 1992.

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

[10] L. Giuri and P. Iglio. Role templates for content-based access control. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 153–159, Nov. 1997.

[11] A. Kern. Advanced features for enterprise-wide role-based access control. In *Proceedings of the 18th Annual Computer Security Applications Conference*, pages 333–343, Dec. 2002.

[12] A. Kern, A. Schaad, and J. Moffett. An administration concept for the enterprise role-based access control model. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, pages 3–11, June 2003.

[13] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.

[14] A. D. Marshall. A financial institution's legacy mainframe access control system in light of the proposed NIST RBAC standard. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC 2002)*, pages 382–390, 2002.

[15] J. D. Moffett. Control principles and role hierarchies. In *Proceedings of the Third ACM Workshop on Role-Based Access Control (RBAC 1998)*, Oct. 1998.

[16] J. D. Moffett and E. C. Lupu. The uses of role hierarchies in access control. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC 1999)*, Oct. 1999.

[17] NSA. Security enhanced linux. http://www.nsa.gov/selinux/.

[18] M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security*, 2(1):3–33, Feb. 1999.

[19] S. Oh and R. S. Sandhu. A model for role admininstration using organization structure. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, June 2002.

[20] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[21] R. S. Sandhu and V. Bhamidipati. Role-based administration of user-role assignment: The URA97 model and its Oracle implementation. *Journal of Computer Security*, 7, 1999.

[22] R. S. Sandhu, V. Bhamidipati, E. Coyne, S. Ganta, and C. Youman. The ARBAC97 model for role-based administration of roles: preliminary description and outline. In *Proceedings of the Second ACM workshop on Role-based access control (RBAC 1997)*, pages 41–50, Nov. 1997.

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

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

[25] R. S. Sandhu and Q. Munawer. The ARBAC99 model for administration of roles. In *Proceedings of the 18th Annual Computer Security Applications Conference*, pages 229–238, Dec. 1999.

[26] A. Schaad, J. Moffett, and J. Jacob. The role-based access control system of a European bank: A case study and discussion. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, pages 3–9. ACM Press, 2001.

[27] T. C. Ting. A user-role based data security approach. In C. Landwehr, editor, *Database Security: Status and Prospects. Results of the IFIP WG 11.3 Initial Meeting*, pages 187–208. North-Holland, 1988.

[28] H. Wang and S. L. Osborn. An administrative model for role graphs. In *Proceedings of the 17th Annual IFIP WG11.3 Working Conference on Database Security*, Aug. 2003.

[29] H. F. Wedde and M. Lischka. Cooperative role-based administration. In *Proceedings of the Eighth ACM Symposium on Access control models and technologies (SACMAT 2003)*, pages 21–32. ACM Press, June 2003.

[30] H. F. Wedde and M. Lischka. Modular authorization and administration. *ACM Transactions on Information and System Security (TISSEC)*, 7(3):363–391, Aug. 2004.