CERIAS Tech Report 2005-29

A CRITIQUE OF THE ANSI STANDARD ON ROLE BASED ACCESS CONTROL

by Ninghui Li and Ji-Won Byun and Elisa Bertino

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

A Critique of the ANSI Standard on Role Based Access Control

Ninghui Li Ji-Won Byun Elisa Bertino
CERIAS and Department of Computer Science
Purdue University
656 Oval Drive, West Lafayette, IN 47907-2086
{ninghui,byunj,bertino}@cs.purdue.edu

Abstract

The American National Standard Institute (ANSI) Standard on Role-Based Access Control (RBAC) was approved in 2004 to fulfil "a need among government and industry purchasers of information technology products for a consistent and uniform definition of role based access control (RBAC) features". While the ANSI RBAC standard represents an important development in RBAC research, it nonetheless has limitations, design flaws, and technical errors. In this paper, we identify the issues in the current ANSI RBAC standard and suggest how they can be fixed. We also present an alternative RBAC framework that is free of the problems that we have uncovered in the standard.

1 Introduction

Role Based Access Control (RBAC) [2, 11, 12, 13, 14, 39] is today's dominant access control paradigm. The past decade has seen an explosion of research in RBAC. Hundreds of papers have been written on topics related to RBAC. The industry's interest in RBAC has also increased dramatically, with most major information technology vendors offering products that incorporate some form of RBAC. Today, all major DBMS products support RBAC. In Windows Server 2003, Microsoft introduced Authorization Manager, which brings RBAC to the Windows operating systems. RBAC has also been used in Enterprise Security Management Systems such as IBM Tivoli Policy Manager [18] and SAM Jupiter [3, 20, 21, 22].

The American National Standard Institute (ANSI) RBAC Standard was approved in 2004 to fulfil "a need among government and industry purchasers of information technology products for a consistent and uniform definition of role based access control (RBAC) features" [2]. The rationale for developing such a standard is explained in the foreword of the standard [2]:

> In recent years, vendors have begun implementing role based access control features in their database management systems, security management and network operating system products, without general agreement on the definition of RBAC features. This lack of a widely accepted model results in uncertainty and confusion about RBAC's utility and meaning. This standard seeks to resolve this situation by using a reference model to define RBAC features and then describing the functional specifications for those features.

The standard has gone through several rounds of open public review. An initial draft of the standard [33] was proposed at the 2000 ACM Workshop on RBAC. A panel was held at the ACM Workshop to discuss the document, and comments have been published in the workshop proceedings [16]. The second version appeared in ACM Transactions on Information and Systems Security (TISSEC) in 2001 [14] and was then submitted to the InterNational Committee for Information Technology Standards (INCITS) in October 2001.

The final version was approved in February 2004 as the American National Standard ANSI INCITS 359-2004. Plans are underway to improve the standard and move the standard to ISO - International Organization for Standardization.

The RBAC standard consists of two parts: the ***Reference Model*** and the ***System and Administrative Functional Specification*** (***Functional Specification*** for short). The Reference Model defines sets of basic RBAC elements and relations that are included in the standard. The Reference Model intends to serve two purposes. One is to rigorously define the scope of RBAC features that are included in the standard; the other is to provide a precise language for defining the Functional Specification, which specifies the operations and functions an RBAC system should support. The RBAC standard includes four components: Core RBAC, Hierarchical RBAC, Static Separation of Duty (SSD) Relations and Dynamic Separation of Duty (DSD) Relations. These components group related features together. Both the Reference Model and the Functional Specification are divided into four parts corresponding to the four components.

While the ANSI RBAC standard represents an important development in RBAC research, it nonetheless has limitations, design flaws, and technical errors. Many of these shortcomings seem to have been overlooked over the development life-cycle of the standard. Some of the most important issues with the standard that we discuss in this paper include:

- The Core RBAC component includes the notion of sessions, which is not essential to RBAC and does not exist in many important RBAC-based security products. As a result, these products cannot be said to use RBAC according to the standard. Similarly, the standard does not accommodate the design that only one role can be activated in a session, which is used in some existing products.
- The Hierarchical RBAC component defines the inheritance relation to be a partial order, which we show is inappropriate. Although using a partial order to represent role hierarchy has been widely accepted in most RBAC literature, it has a significant weakness when one considers updating the role hierarchy.
- There are several possible interpretations of a role hierarchy, and they interact with constraints in important ways. The standard fails to explain these interactions.
- There are a number of errors in the standard; some are typos while others are more serious technical errors. For example, an obvious mistake is that authorized_permissions(r) is defined to be {u   PRMS | r   r, (p, r)   PA}, whereas r   r should be r   r. A list of these errors is given in Appendix A.
- The Functional Specification also has a number of problems. Some functions seem to be redundant; and some functions seem to be missing. Furthermore, important details are sometimes overlooked. The errors found in the Functional Specification are identified in Appendix C.

The contributions of this paper are as follows.

- We identify a number of technical errors and limitations in the ANSI RBAC standard and suggest how they can be fixed. Almost all of these problems also exist in a widely cited previous version of the standard that appeared in ACM TISSEC in August 2001 [14].
- We show that, to maintain a role hierarchy, one should maintain the role dominance relationships that have been explicitly added and distinguish them from the derived relationships.
- We clarify three interpretations of role hierarchy: user inheritance, permission inheritance and activation inheritance. We discuss their relative benefits and limitations, especially in their interaction with other RBAC features such as constraints.
- We present a new RBAC framework that is inspired by the ANSI RBAC standard and is free of the problems discussed in this paper. We expect this to result in a revised version of the ANSI RBAC standard and to influence the development of an international standard on RBAC.

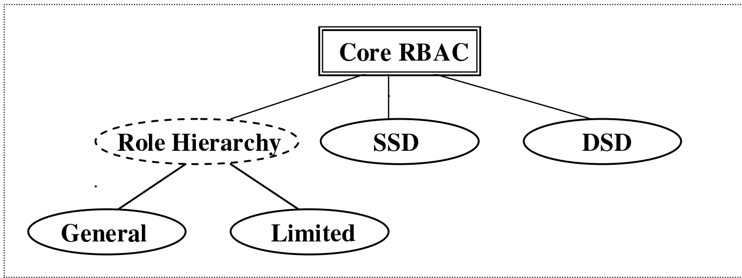


Figure 1: Standard RBAC components and the dependencies

The remainder of this paper is organized as follows. We provide a summary of the current ANSI RBAC Standard in Section 2. In Section 3 we discuss various issues in the current standard and make suggestions for changes. Our new RBAC framework is presented in Section 4. We survey related work in Section 5 and conclude in Section 6.

2 Overview of the ANSI RBAC Standard

Below we provide the specifications of the four components in the ANSI RBAC standard. Figure 1 shows these components and the dependencies among them. As shown, Core RBAC is required in any RBAC system. A particular RBAC system may include any combination of role hierarchy, SSD, and DSD. To include role hierarchy, a system should use either a general or a limited hierarchy, but not both. For a more detailed description, the reader is directed to the standard [2] (or the previous version [14]).

Core RBAC The basic concept of RBAC is that permissions are assigned to roles and individual users obtain such permissions by being assigned to roles. Core RBAC captures this basic concept. The Core RBAC component in the Reference Model includes the following sets, functions and relations, which are taken verbatim from [2].

- USERS, ROLES, OPS, and OBS (users, roles, operations and objects respectively).
- UA   USERS × ROLES, a many-to-many mapping user-to-role assignment relation.
- assigned_users : (r : ROLES) 2^{USERS}, the mapping of role r onto a set of users. Formally: assigned_users(r) = {u   USERS | (u,r)   UA}
- PRMS = $2^{(OPS \times OBS)}$, the set of permissions.
- PA   PRMS × ROLES, a many-to-many mapping permission-to-role assignment relation.
- assigned_permissions(r : ROLES) 2^{PRMS}, the mapping of role r onto a set of permissions. Formally: assigned_permission(r) = {p   PRMS | (p,r)   PA}
- Op(p : PRMS) { op   OPS}, the permission to operation mapping, which gives the set of operations associated with permission p.
- Ob(p : PRMS) { op   OBS}, the permission to object mapping, which gives the set of objects associated with permission p.
- SESSIONS = the set of sessions
- session_users(s : SESSIONS)   USERS, the mapping of session s onto the corresponding user.
- session_roles(s : SESSIONS) 2^{ROLES}, the mapping of session s onto a set of roles. Formally: session_roles(s_i) { r   ROLES | (session_users(s_i), r)   UA}

- avail_session_perms(s : SESSIONS) 2^{PRMS}, the permissions available to a user in a session = $_{r \ \ session\_roles(s)}$ assigned_permissions(r)

Hierarchical RBAC The Hierarchical RBAC component introduces role hierarchies, which define an inheritance relation among roles in order to reduce the cost of administration. The Hierarchical RBAC component includes two types of role hierarchies: general role hierarchies and limited role hierarchies. Below are discussions and specifications for Hierarchical RBAC, taken verbatim from the standard [2, 14]. We use footnotes to point out four errors in them.

> Role hierarchies define an inheritance relation among roles. Inheritance has been described in terms of permissions; i.e., r_1 inherits r_2 if all privileges of r_2 are also privileges of r_1. ... [1]
>
> General Role Hierarchies
>
> - RH ROLES × ROLES is a partial order on ROLES called the inheritance relation, written as , where r_1 r_2 only if all permissions of r_2 are also permissions of r_1, and all users of r_1 are also users of r_2, i.e., r_1 r_2 authorized_permissions(r_2) authorized_permissions(r_1).
> - authorized_users(r : ROLES) 2^{USERS}, the mapping of role r onto a set of users in the presence of a role hierarchy. Formally: authorized_users(r) = {u USERS | r r, (u, r) UA}
> - authorized_permissions(r : ROLES) 2^{PRMS}, the mapping of role r onto a set of permissions in the presence of a role hierarchy. Formally: authorized_permissions(r) = {p PRMS | r r, (p, r) PA} [2]
>
> Node r_1 is represented as an immediate descendant of r_2 by r_1 r_2, if r_1 r_2, but no role in the role hierarchy lies between r_1 and r_2. That is, there exists no role r_3 in the role hierarchy such that r_1 r_3 r_2, where $r_1 = r_2$ and $r_2 = r_3$. [3]
>
> Limited Role Hierarchies
>
> - General Role Hierarchies with the following limitation:
>   r, r_1, r_2 ROLES, (r r_1 r r_2) ($r_1 = r_2$). [4]

A limited role hierarchy forms a forest of inverted trees. In other words, there are a number of junior-most roles (i.e., the roots of these inverted trees), and any of the other roles has a single immediate descendant. As discussed in [33], an inverted tree facilitates sharing of resources. Resources made available to a junior-most role are also available to other more senior roles. However, an inverted tree does not allow aggregation of resources from more than one role.

Constrained RBAC The Constrained RBAC component contains two types of separation of duty relations: Static Separation of Duty (SSD) and Dynamic Separation of Duty (DSD). An SSD constraint is specified by a role set rs such that $|rs| \geq 2$ and a cardinality n such that $2 \leq n \leq |rs|$; it means that no user can be authorized for n or more roles in rs. Like SSD, a DSD constraint is specified by a role set rs such that $|rs| \geq 2$ and a cardinality n such that $2 \leq n \leq |rs|$; it means that no user may simultaneously activate n

[1] This suggests that the role hierarchy is inferred from the privileges the roles have, which is incorrect. If r_1 and r_2 are independently assigned the same permissions, r_1 does not have to inherit r_2, nor does r_2 have to inherit r_1.

[2] r r should be r r.

[3] The condition $r_1 = r_2$ should be $r_1 = r_3$.

[4] The definition is incorrect as it effectively limits the maximum height of role hierarchies to be two. To see this, observe that if r r_1 r_2, then the condition requires that $r_1 = r_2$. To correctly define the limitation, (r r_1 r r_2) should be (r r_1 r r_2).

or more roles from rs in one session. The difference between SSD and DSD is that while a SSD constraint limits the permissions for which a user can be authorized, a DSD constraint limits the permissions that a user can use in one session. The followings are taken verbatim from the standard.

Static Separation of Duty

- $(rs, n) \quad SSD, \; t \quad rs : |t| \geq n \qquad _{r \; t} \, assigned\_users(r) = \quad .$

Static Separation of Duty in the Presence of a Hierarchy

- $(rs, n) \quad SSD, \; t \quad rs : |t| \geq n \qquad _{r \; t} \, authorized\_users(r) = \quad .$

Dynamic Separation of Duty

- $rs \quad 2^{ROLES}, n \quad N, (rs, n) \quad DSD \qquad n \geq 2, |rs| \geq n$, and
  $s \quad SESSIONS, \; rs \quad 2^{ROLES}, \; role\_subset \quad 2^{ROLES}, \; n \quad N, (rs, n) \quad DSD,$
  $role\_subset \quad rs, role\_subset \quad session\_roles(s) \quad | \quad role\_subset| < n.$

3 Issues in the ANSI RBAC Standard

In this section, we make eight suggestions on changes to the current RBAC standard. We discuss the rationale underlying these suggestions by discussing the issues we have identified from the standard.

Suggestion 1 ***The notion of sessions should be removed from Core RBAC and introduced in a separate component.***

The Core RBAC component includes the notion of sessions, where a session is defined as "a mapping between a user and an activated subset of roles that are assigned to the user" [2]. We argue that the notion of sessions should not be included in Core RBAC; instead, it should be included in a new optional component.

While the notion of sessions is very useful in some applications (such as DBMS), it is not applicable in some other applications. For example, in Enterprise Security Management (ESM) systems such as SAM Jupiter [21, 20, 22], IBM Tivoli [19], and the Role Control Center [11], RBAC is used to provide the central management for authorizations over a number of heterogeneous target systems (e.g., operating systems, applications, and databases). Note that ESM systems are not Single-Sign-On systems. In these ESM systems, users are assigned memberships in roles and gain permissions on abstract representations of the physical resources in the target systems. Then the ESM systems change the policy settings in target systems (e.g., via creating new accounts, changing group memberships of accounts, and changing access control lists) to provide users authorizations in the target systems. Users interact directly with the target systems to access resources; the ESM products only use RBAC to manage the policy settings in the target systems. The notion of sessions does not exist in such systems as permission usages happen in target systems and are outside the ESM systems.

The RBAC standard mandates: "Not all RBAC features are appropriate for all applications. As such, this standard provides a method of packaging features through the selection of functional components and feature options within a component, beginning with a core set of RBAC features that must be included in all packages." According to the above statement, ESM products such as SAM Jupiter [21, 20, 22], IBM Tivoli [19], and the Role Control Center [11] do not use RBAC. However, it has been widely agreed that these ESM products are among the most important applications of RBAC. Also, the prospect of using these ESM products to greatly reduce administrative cost has been used as one of the strongest justifications for RBAC [28]. Furthermore, these products often drive the research on RBAC.

By including the notion of sessions in Core RBAC, the current standard unnecessarily restricts RBAC. The basic concept of RBAC is that permissions are assigned to roles, and users obtain such permissions by being assigned to roles. This simple concept, with or without features such as sessions, has been demonstrated to provide powerful and useful access control systems. Therefore, we argue that the notion of sessions should be included in a component other than Core RBAC.

Suggestion 2 ***The standard should accommodate RBAC systems that allow only one role to be activated in a session.***

In the standard, multiple roles can be activated in one session. However, some RBAC systems (e.g., that in Baldwin [4] and in Informix according to [31]), only one role can be activated in a session. Therefore, one cannot say that such systems implement RBAC with sessions according to the standard. We now argue that the standard should accommodate these systems. We compare the following two approaches.

Single-role activation (SRA) Only one role can be activated in a session.

Multi-role activation (MRA) Multiple roles can be activated in one session, and DSD constraints may be used to restrict concurrent activation of some roles.

One can argue that SRA is sometimes more desirable. Consider a situation in which a user is assigned to both the Quality-Assurance role and the Developer role but is not allowed to use both roles at the same time in one session. The SRA design automatically ensures that only one of these roles can be activated in any session. In MRA, this has to be achieved with DSD constraints which add significant complexity. Further observe that if one wants to allow a user to use permissions of several roles in one session, one can define a new role that dominates all these roles and allow the user to activate this new role. The difference between SRA and MRA is that in SRA one has to do extra work to enable more accesses (by creating new roles) while in MRA one has to do extra work to restrict access (by adding constraints). Therefore, SRA is better than MRA not only because it is simpler but also because it better achieves the ***fail-safe defaults*** principle identified in [32]. The following is quoted from [32].

> Fail-safe defaults: Base access decisions on permission rather than exclusion. This principle, suggested by E. Glaser in 1965 means that the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted. The alternative, in which mechanisms attempt to identify conditions under which access should be refused, presents the wrong psychological base for secure system design. A conservative design must be based on arguments why objects should be accessible, rather than why they should not. In a large system some objects will be inadequately considered, so a default of lack of permission is safer. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situation, since it will be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use.

Thus, we argue that an RBAC standard should accommodate SRA. In fact, we would suggest that, in any RBAC implementation that needs to use sessions, the tradeoff between SRA and MRA should be considered.

Suggestion 3 ***Derived (and thus redundant) functions should be removed from the Reference Model.***

The specification of the Reference Model does not clearly distinguish base relations and derived functions. For example, the Core RBAC specification includes both $UA \subseteq USERS \times ROLES$ and $assigned\_users : (r : ROLES) \rightarrow 2^{USERS}$. Each of the two can be derived from the other. In fact, the standard defines assigned_users in terms of UA as follows: $assigned\_users(r) = \{u \in USERS \mid (u,r) \in UA\}$, which suggests that the function assigned_users is derived from the relation UA. We believe that only one of them should be listed in Core RBAC, for the reasons discussed below.

By listing both UA and assigned_users in the Reference Model, the administrative functions (e.g., ***AssignUser***, ***DeassignUser*** and ***DeleteUser***) must modify both relations and maintain their consistency. In fact, the way these functions are defined in the Functional Specification indicates that UA and assigned_users are maintained independently; i.e., invoking an administrative function will result in updates to both relations. This unnecessarily complicates the specification of the administrative functions. Furthermore, the review functions for Core RBAC include the ***AssignedUser*** function, which achieves exactly the same effect as assigned_users and is defined in terms of UA. It is not clear what benefit assigned_users brings. Finally, as the standard includes both UA and assigned_users, it is unclear why assigned_roles : (u : USERS) 2^{ROLES} is omitted.

Other redundant functions in Core RBAC include assigned_permissions, which is derived from the permission assignment relation PA. The relations and functions in Core RBAC that deal with sessions also contain a redundant function: avail_session_perms, which is derived from session_roles and PA.

In summary, we suggest that derived functions such as assigned_users, assigned_permissions, and avail_session_perms be removed from the Reference Model and defined only as review functions.

Suggestion 4 ***The Reference Model should maintain a relation that contains the role dominance relationships that have been explicitly added, and update this relation when the role hierarchy changes.***

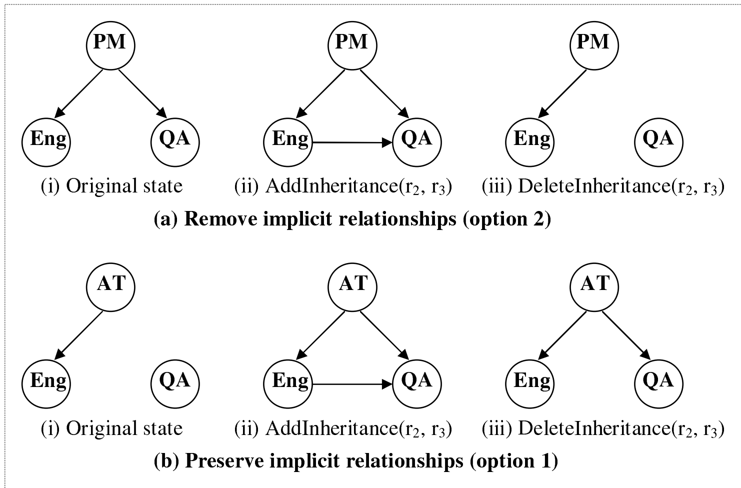
In the Hierarchical RBAC component, a relation RH is used and is assumed to be a partial order. (See Appendix B for terminologies on binary relations.) While the treatment of RH as a partial order has been standard in the literature on RBAC (e.g., in the influential RBAC96 models [39] and many other papers on RBAC), we argue that this is inappropriate when updates on the role hierarchy are considered. We suggest that RH include only the role dominance relationships that have been explicitly added and that RH be an ***irreflexive*** and ***acyclic*** relation. Changes to the role hierarchy are carried out by changes to RH. A derived relation is then defined to be the partial order entailed by RH, i.e., the reflexive and transitive closure of RH. We now discuss the rationale for our suggestion. In the standard, the following administrative functions are defined (we use a slightly different notation to improve readability):

AddInheritance(r_asc, r_desc)
  if ¬(r_asc r_desc) ¬ (r_desc r_asc)
  then = { (r,q) | r r_asc r_desc q }
DeleteInheritance(r_asc, r_desc)
  if (r_asc r_desc)
  then = (\ { (r_asc, r_desc) })

in which denotes the role hierarchy partial order before the change, denotes the relation after the change, denotes the immediate predecessor relation before the change, denotes the relation after the change, \ is the set difference operator, and ( ) is the reflexive and transitive closure operator. Recall that r_1 r_2 if r_1 r_2 and there exists no role r_3 such that r_1 r_3 r_2, $r_1 = r_3$, and $r_2 = r_3$.

The problem with the above definitions is that after adding and deleting a role in a role hierarchy, one may not be able to return to the original state. For instance, consider the RBAC state in Figure 2(a)(i), which includes the following role dominance relationships: ProjManager Engineer and ProjManager QA. Suppose that when a product is about to be released, one wants the engineers to also serve as QAs, so one adds a temporary relationship Engineer QA, resulting in the role hierarchy in Figure 2(a)(ii). After the release, one wants to delete the temporary relationship, expecting the hierarchy to return to the original state in Figure 2(a)(i). However, using DeleteInheritance in the standard, the relationship ProjManager QA will also be deleted, resulting in the role hierarchy in Figure 2(a)(iii).

Some authors suggested that one should keep all other role dominance relationships while removing one, e.g., in the administrative model for RBAC proposed in [10]. Using this interpretation, ProjManager QA is maintained after deleting Engineer QA. However, this introduces other problems. Consider the RBAC state in Figure 2(b)(i), which contains the following relationships: Architect Engineer. After adding



ProjManager (PM), Engineer (Eng), Quality Assurance (QA), and Architect (AT)

Figure 2: Adding and deleting a role from RH

Engineer QA, the state changes to Figure 2(b)(ii). After removing Engineer QA, one would expect 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, the resulting state would be Figure 2(b)(iii), which is undesirable.

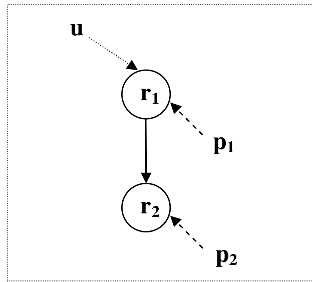
In fact, the standard acknowledges that the two options exist and includes the following:

> When DeleteInheritance is invoked with two given roles, say Role A and Role B, the implementation system is required to do one of two things: (1) The system may preserve the implicit inheritance relationships that roles A and B have with other roles in the hierarchy. That is, if role A inherits other roles, say C and D, through role B, role A will maintain permissions for C and D after the relationship with role B is deleted; (2) A second option is to break those relationships because an inheritance relationship no longer exists between Role A and Role B. The question of which semantics the DeleteInheritance is left as an implementation issue and is not prescribed in this specification.

Observe that the above discussion is inconsistent with the definition of DeleteInheritance in the standard, which adopts the second option. Furthermore, as previously discussed, neither option is satisfactory. As neither option is "more correct" than the other, one should not be forced to choose one or the other. The problem lies in the fact that, maintaining only a partial order, one cannot distinguish those role dominance relationships that have been explicitly added from those that are implied. In other words, the partial order derived from the explicitly added role dominance relationships contains less information than the role dominance relationships. For example, two different sets of role dominance relationships may entail exactly the same partial order. From the partial order, one cannot tell which set is the intended one. Maintaining only the derived partial order means that one does not maintain enough information about the current RBAC state and problems arise when changes to the role dominance relationships are made.

The solution we propose is to maintain explicitly added role dominance relationships in RH and use it to derive the implied partial order . For performance considerations, an RBAC system could choose to cache , as long as it can tell which dominance relationship was explicitly added and which was derived.

We emphasize that this issue should not be considered a minor implementation detail. Administration of RBAC is an open problem that is being actively researched [10, 27, 29, 38, 36, 40], and a consensus has



$UA = \{(u, r_1)\}, PA = \{(r_1, p_1), (r_2, p_2)\}, RH = \{r_1 \succeq r_2\}$

Figure 3: An RBAC state

yet to be reached. One key question, which has been overlooked so far, is how a role hierarchy should be maintained. When an RBAC paper mentions a role hierarchy, it almost always treats it as a partial order. This is probably because most researchers are familiar with Mandatory Access Control (MAC) [5], where security levels are organized as a lattice (which is a partial order), and immediately make an association between partial orders and role hierarchies. As we argue above, the dynamic nature of role hierarchies (as opposed to the fixed security level lattices) requires a different approach.

Suggestion 5 ***The semantics of role inheritance should be clearly specified and discussed.***

There are three possible interpretations for a role hierarchy; a particular RBAC system may choose to implement one or more of these interpretations. For example, consider the following situation illustrated in Figure 3: $UA = \{(u, r_1)\}$, $PA = \{(r_1, p_1), (r_2, p_2)\}$, and $RH = \{r_1 \succeq r_2\}$. That $r_1 \succeq r_2$ may mean one or more of the following:

1. User Inheritance (UI): All users that are authorized for the role r_1 are also authorized for the role r_2. The user u is authorized for the role r_2 and is therefore authorized for the permission p_2. However, under this interpretation alone, the role r_1 is not authorized for the permission p_2.
2. Permission Inheritance (PI): The role r_1 is automatically authorized for all permissions for which the role r_2 is authorized. Under this interpretation alone, u is authorized for r_1 but not for r_2; however, u is nonetheless authorized for the permission p_2 as r_1 is authorized for p_2.
3. Activation Inheritance (AI): When r_1 is activated in a session, r_2 is also activated in the session. This interpretation makes sense only when MRA sessions are used, i.e., $\neg MRA \Rightarrow \neg AI$. Under this interpretation alone, u cannot activate r_2 directly; however, u can activate r_1, indirectly causing r_2 to be activated. In other words, u cannot use permission p_2 in a session without activating r_1.

We point out that all three kinds of inheritance semantics have been mentioned or alluded to in the standard. However, a clear specification and discussion of their relationships and interactions with other features in the standard are missing, and the standard is sometimes inconsistent about which semantics should be used. Sandhu [35] discussed the permission-usage aspect of role hierarchies, which corresponds to PI, and the role-activation aspect of role hierarchies, which corresponds to UI. AI is not discussed in [35]. We also note that UI and AI have been implemented in Oracle [30].

When there are no sessions or constraints, UI and PI have exactly the same effect, as the only thing that matters in such systems is the set of permissions for which a user is authorized. These three interpretations differ when there are sessions or constraints.

- When there are (SRA or MRA) sessions, under UI alone, u can use p_2 only if r_2 is explicitly activated by u. Under PI alone, u activates r_1 to use permissions p_1 and p_2, but u cannot activate r_2. With SRA, only a single role can be activated in a session; thus AI cannot be used, and the only way to allow u to use both p_1 and p_2 is to use PI. With MRA, the effects of PI and AI are similar; they differ when there are also DSD constraints.

UI makes it easier to achieve the least privilege principle, as a user can activate a less powerful role when that is sufficient for the current task. On the other hand, PI or AI may be considered to be more user friendly, as u can use the role r_1 to have both p_1 and p_2 without knowing about the existence of r_2. In other words, the intricate details of how permissions are set up through roles can be partially hidden from a user. Without either PI or AI, the user u has to know r_2 and explicitly activate r_2 in order to use p_2. Therefore, it seems desirable to have UI and at least one of PI and AI in such systems. We summarize this as MRA (UI (PI AI)) and SRA (UI PI ¬ AI).

- When there are SSD constraints, UI seems to be necessary. With just PI and not UI, the intention of SSD constraints can be circumvented. For example, if two roles r_1 and r_2 are declared to be mutually exclusive, the intention is that no user should be authorized for the combined permissions of r_1 and r_2. However, with just PI and not UI, one can define a role r_3 to dominate both r_1 and r_2 and assign a user u to r_3 without violating the constraint, as u is not authorized for r_1 or r_2 without UI. When sessions exist and AI is used, a similar argument can be used to infer that UI should also be used. We summarize this as SSD ((PI AI) UI). As at least one of the three must be used, this implies SSD UI.
- DSD constraints only make sense when MRA sessions exist. With DSD constraints, it is undesirable to have PI but not AI for reasons similar to the above. For example, suppose that two roles r_1 and r_2 are declared to be dynamically mutually exclusive and that r_1 r_2. With PI but not AI, a user can exercise the combined permissions from both r_1 and r_2 without violating the constraint, as the user can use the permissions of r_2 without activating it. Therefore, when DSD constraints exist, PI must be used together with AI. We summarize this as DSD (PI AI).

The RBAC standard adopts UI and PI, but not AI. In Section A.2.2, the standard reads "When that given role is activated by a user, the question of whether the inherited roles are automatically activated or must be explicitly activated by a user is left as an implementation issue and no one course of action is prescribed as part of this specification." However, from the ways functions such as AddActiveRole are defined, one can infer that the Functional Specification adopts the "no AI" approach. The AddActiveRole function adds only the role that has been explicitly specified to the session_roles relation, and the check for DSD constraints checks only the roles in session_roles. As discussed above, this is undesirable as the effect of DSD constraints can be circumvented.

Our suggestion is to specify and discuss the three interpretations for role hierarchies and to define the Functional Specification based on one recommended combination. One combination that is consistent with our analysis is to implement all the interpretations that apply, that is, to always use both UI and PI and to add AI when there are MRA. The standard should probably allow products to implement other combinations; however, such deviation should be justified and documented.

Suggestion 6 ***Interaction between role hierarchies and SSD constraints should be discussed.***

The standard says "Core RBAC is required in any RBAC system, but the other components [i.e., role hierarchies, SSD constraints and DSD constraints] are independent of each other and may be implemented separately". As previously discussed, the interpretations of role hierarchies interact with constraints in important ways. There are other interactions as well. A set of SSD constraints may be incompatible with a role hierarchy, in the following sense. A set of SSD constraints may preclude us from assigning any user to some roles in RH. For example, if $\{(r_3 \geq r_1), (r_3 \geq r_2)\}$ RH, then the constraint that r_1 and r_2 are mutually exclusive implies that no user is allowed to be authorized for r_3 (under the UI interpretation). This means that no user can ever be assigned to r_3 or any role that dominates r_3; as such the role r_3 seems useless. How to deal with such an incompatibility between a role hierarchy and SSD constraints should be discussed in the standard. One approach is to disallow such incompatibility, as such incompatibility may signify an error in the design of the policy.

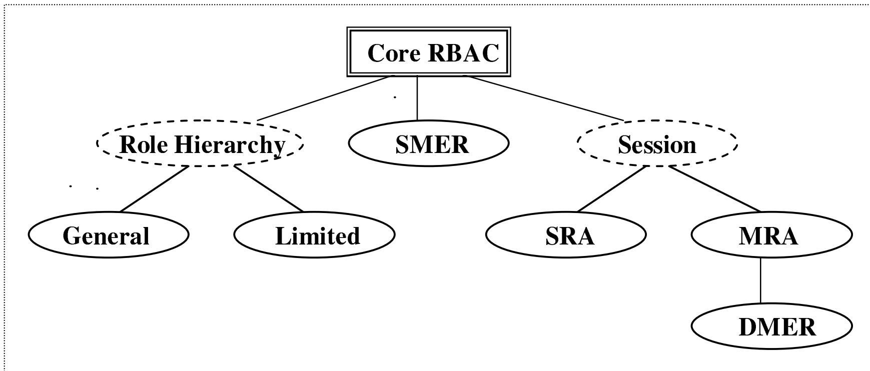
Suggestion 7 ***More accurate terminologies for constraints (i.e., SSD and DSD) should be adopted to avoid any misinterpretation.***

The standard uses Static Separation of Duty (SSD) and Dynamic Separation of Duty (DSD) to represent mutually exclusive role constraints. However, as discussed by Li et al. [24], these terminologies do not accurately describe the effects of such constraints and can be misleading as they blur the distinction between objectives and mechanisms. What are referred to as SSD constraints are only mechanisms that may be applied to enforce Separation of Duty (SoD) policies. What are referred to as DSD constraints actually do not enforce SoD policies; instead, they are motivated by the least privilege principle. Li et al. [24] propose to call them ***Static Mutually Exclusive Roles (SMER)*** and ***Dynamic Mutually Exclusive Roles (DMER)*** constraints. We now reproduce their rationale here.

The concept of SoD has long existed in the physical world, sometimes under the name "the two-man rule". SoD has also been recognized as one of the fundamental principles in computer security [6, 32], as it ensures that "no single accident, deception, or breach of trust is sufficient to compromise the protected information" [32]. For example, an SoD policy may require the cooperation of at least k (for some $k \geq 2$) different users to complete a sensitive task. In static enforcement, a Static SoD (SSoD) policy requires that no $k-1$ users together have all permissions to complete a sensitive task. In RBAC, SMER constraints are commonly used to implement such policies. A SMER constraint requires that no user is a member of t or more roles in a set of m roles $\{r_1, r_2, \cdots, r_m\}$. Whether a set of SMER constraints implement any SSoD policy clearly depends on how permissions are assigned to roles. If all permissions are assigned to one role, then SMER constraints cannot enforce any SoD policies. SSoD policies are ***objectives*** that need to be achieved, and SMER constraints are ***mechanisms*** used to achieve SSoD policies, specifically in RBAC. However, in most RBAC literature, this distinction between objectives and mechanisms has not been clearly made. As a result, the standard also adopts the term SSD to refer to SMER constraints. One danger of this terminology, which implicitly equates SMER constraints with SSoD policies, is that one may set up SMER constraints and falsely believe that the SSoD policies are correctly enforced; however, when the permission assignment changes, the SMER constraints may no longer be adequate for enforcing the intended SSoD policies.

DMER constraints limit the roles a user can activate in a single session. They are introduced in the standard under the name DSD constraints, presumably because they are the "dynamic" version of the so-called "SSD constraints" (which in our opinion should be called SMER constraints). However, DMER constraints do not seem to enforce SoD policies at all because they do not prevent a sensitive task from being completed by a single user. For example, suppose that two roles r_1 and r_2 are declared to be dynamically mutually exclusive in a DMER constraint; presumably because in order to complete a sensitive task, one has to combine permissions assigned to r_1 with permissions assigned to r_2. As each session can have only one user, this task cannot be finished in any single session, and multiple sessions are needed to complete the task. A user can thus start a session, activate r_1, use the permissions of r_1 to work on the task, end the session, start another session, activate r_2, and use the permissions of r_2 to finish the task. This does not violate the DMER constraint, but clearly violates the intended SoD policy. In fact, DMER constraints are motivated by the least privilege principle, which mandates that "every program and every user of the system should operate using the least set of privileges necessary to complete the job" [32]. By requiring certain roles to be not activated at the same time, one can limit the privileges that a user may use in a session.

Thus, our suggestion is to adopt more appropriate terms for SSD and DSD; for example, SMER and DMER. Also, the standard should note that DMER constraints are suitable to enforce the least privilege principle rather than the separation of duty principle.



SMER: Statically Mutually Exclusive Role constraint, DMER: Dynamically Mutually Exclusive Role constraint, SRA: Single-Role Activation, MRA: Multi-Role Activation

Figure 4: RBAC Components and Dependencies

Suggestion 8 ***All technical errors should be corrected.***

The standard contains a number of minor errors. Some are typos while others are more serious technical mistakes. Needless to say, such errors should not be allowed in a national standard. In Appendix A, we provide a brief summary of the errors we have found in the standard.

The Functional Specification also has a number of problems. Some functions seem to be redundant; and some functions seem to be missing. Furthermore, important details are sometimes overlooked. One example is AddActiveRole, which is a supporting system function defined for General Role Hierarchies. This function first ensures that the user is indeed authorized for the role to be added and then adds the role to the relation session_roles; thus the relation session_roles contains only the roles that are explicitly activated and does not contain other roles that are dominated by the activated roles. This could be a reasonable approach, provided that those dominated roles are considered whenever necessary, e.g., in CheckAccess. However, CheckAccess, only defined for Core RBAC and assumed to be valid for other components, uses only the permissions that are explicitly assigned to session_roles. In other words, the current Functional Specification does not implement either PI or AI. This seems to be inconsistent as the standard seems to support PI. For example, the review function RolePermissions for General Role Hierarchies clearly implements PI. In order to be consistent, either AddActiveRole or CheckAccess must be redefined for Hierarchical RBAC. The errors found in the Functional Specification are identified in Appendix C.

4 A New RBAC Framework

Based on our analysis of the ANSI RBAC standard in the last section, we propose a new framework for RBAC. Components of the framework are illustrated in Figure 4. Core RBAC identifies the minimum set of features that an RBAC system should include. Role hierarchy, SMER, Session, and DMER include more advanced RBAC features. Core RBAC is required for any RBAC system. An RBAC system that implements role hierarchy should implement either a general role hierarchy or a limited role hierarchy. An RBAC system that implement sessions should use either SRA sessions or MRA sessions. DMER can be included only if MRA session is also included in an RBAC system.

Following the ANSI standard, our RBAC framework consists of a ***Reference Model*** and a ***Functional Specification***. The Reference Model is described below.

Core RBAC   An RBAC system should (explicitly or implicitly) identify the following universal sets. These (potentially infinite) sets include those objects that exist in the RBAC system and those objects that could be added. These sets serve as data types for functions such as adding a new user and adding a new role.

- U: the set of all possible users. For example, if each user is identified by an account name, then U consists of all strings that could be used as an account name.
- R: the set of all possible roles.
- P: the set of all possible permissions.

An RBAC system should maintain the following sets and relations as the state of the system:

- USERS U : the set of users currently in the system.
- ROLES R : the set of roles currently in the system.
- PRMS P : the set of permissions currently in the system.
- UA USERS × ROLES: the user-to-role assignment relation.
- PA ROLES × PRMS: the permission-to-role assignment relation.

Our Core RBAC does not specify the internal structure of permissions, unlike the ANSI RBAC standard, which defines PRMS OPS × OBS. We feel that it is better to model permissions at an abstract level, because permissions are often implementation-dependent, as pointed out by Sandhu in [34]. Also, the way that permissions are defined in the standard could be problematic as certain operations are applicable only to certain types of objects; for example, in database systems, a relation would have quite different operations from a stored procedure.

Hierarchical RBAC An RBAC system with role hierarchies should maintain the following sets and relations in addition to the ones in Core RBAC, depending on the type of role hierarchies:

General Role Hierarchies

- RH ROLES × ROLES that satisfies the condition that RH is irreflexive and acyclic: this contains the role dominance relationships that have been explicitly added.
- A partial order which is the reflexive and transitive closure of RH. An RBAC system may choose to store or to compute it when needed.

Limited Role Hierarchies

- RH ROLES × ROLES that satisfies the conditions that RH is irreflexive and acyclic and (r_1, r_2) RH, (r r_1) (r r_2) $(r_1 = r_2)$, where r r if and only if r r and r (r r r) (r = r): the additional condition restricts role hierarchies to inverted trees.
- A partial order which is the reflexive and transitive closure of RH. An RBAC system may choose to store or to compute it when needed.

There are three semantics for a role hierarchy:

1. User Inheritance (UI): All users authorized for a role r are also authorized for any role r where r r.
2. Permission Inheritance (PI): A role r is authorized for all permissions for which r is authorized where r r.
3. Activation Inheritance (AI): Activating a role r automatically activates the roles r where r r. Note that this semantics can be used only if MRA sessions are used.

A particular RBAC system may choose to implement one or more of these interpretations. We suggest an RBAC system to implement all the interpretations that apply, that is, to always use both UI and PI and to add AI when there are MRA sessions. For RBAC systems that do not implement all applied interpretations, the following are some guidelines (as discussed under Suggestion 5 in Section 3): ¬ MRA ¬ AI, MRA (UI (PI AI)), SRA (UI PI ¬ AI), SMER UI, DMER (PI AI).

Static Constraints (SMER) An RBAC system with statically mutually exclusive roles (SMER) constraints should (explicitly or implicitly) identify the following universal set.

- C: the set of all possible names for SMER constraints.

An RBAC system with SMER constraints should maintain the following set and relation in addition to the ones in Core RBAC:

- SMER (C × 2^{ROLES} × N): the set of 3-tuples (name, role_set, cardinality), each of which represents an existing SMER constraint in the system.

SMER constraints must satisfy the following conditions:

- c_i C $|\{$ (c, rs, t) SMER $| c = c_i\}| \leq 1$; that is, every SMER constraint has a unique name.
- (c, rs, t) SMER, $2 \leq t \leq |rs|$.
- (No role hierarchies) (c, rs, t) SMER u USERS $|\{r \mid (u, r)$ UA$\}$ rs$| < t$; that is, no user is currently assigned to t or more roles from the set rs in each SMER constraint.
- (With role hierarchies) (c, rs, t) SMER u USERS $|\{r \mid (u, r')$ UA r' r$\}$ rs$| < t$; that is, no user is currently authorized for t or more roles from the set rs in each SMER constraint.

Session An RBAC system with sessions should (explicitly or implicitly) identify the following universal set.

- S: the set of all possible session ID's.

An RBAC system with sessions should maintain the following set and relation in addition to the ones in Core RBAC, depending on the limit on role activation:

Single-role Activation (SRA): Only one role can be activated in a session.

- SESSIONS (S × USERS × 2^{ROLES}): the set of 3-tuples (id, user, activated_roles), each of which represents a currently existing session in the system and satisfies the condition |activated_role| ≤ 1.

Multi-role activation (MRA): Multiple roles can be activated in a session.

- SESSIONS (S × USERS × 2^{ROLES}): the set of 3-tuples (name, user, activated_roles), each of which represents a currently existing session in the system.

The relation SESSIONS satisfies the following conditions:

- s_i S, $|\{$(s, u, rs) SESSIONS $| s = s_i\}| \leq 1$; that is, every session has a unique ID.

Dynamic Constraints (DMER) An RBAC system with dynamically mutually exclusive roles (DMER) constraints should (explicitly or implicitly) identify the following universal set.

- D: the set of all possible names for DMER constraints.

An RBAC system with DMER constraints should maintain the following sets and relations in addition to the ones in Core RBAC:

- DMER (D × 2^{ROLES} × N): the set of a 3-tuple (name, role_set, cardinality), each of which represents an existing DMER constraint in the system.

DMER constraints must satisfy the following conditions:

- d_i D, $|\{(d,rs,t)$ DMER $| d = d_i\}| \leq 1$; that is, every existing DMER constraint has a unique name.
- (d,rs,t) DMER, $2 \leq t \leq |rs|$.
- (d,rs,t) DMER (s,u,srs) SESSIONS $|srs$ $rs| < t$; that is, there is no session that t or more roles from the set rs in each DMER constraint are activated.

Functional Specification Functions are divided into two categories: ***Administrative Functions*** and ***Review Functions***. The Administrative functions include the functions that are essential to maintain an RBAC system while the Review functions include the functions that are helpful to assess a particular RBAC state. In other words, the Administrative functions change the current RBAC state, and the Review functions do not. Below we provide a list of major improvements of our functional specification over the one in the ANSI RBAC Standard. The complete version of the Functional Specification is in Appendix C.

- A number of review functions are added to Core RBAC to provide a common interface for RBAC with and without role hierarchies. For instance, the function AuthorizedRoleUsers in Core RBAC returns a set of users that are assigned to a given role. This function is overridden in Hierarchical RBAC and returns a set of users that are authorized for a given role.
- A number of administrative functions are added, for example, functions for introducing or removing permissions in an RBAC system.
- Many errors are fixed. For instance, the function DeleteRole is redefined for each advanced component to make appropriate changes. Also, the functions for activating/deactivating roles, e.g., AddActiveRole and DropActiveRole, are modified to consider inheritance relationships.

5 Related Work

The notion of roles was first introduced to access control in the context of database security [4,43] as a means to group permissions together to ease security administration. The term "Role Based Access Control" was first coined by Ferraiolo et al. [12,13]. Sandhu et al. [39] developed the influential RBAC96 family of RBAC models, which consists of four sub-models. $RBAC_0$ is equivalent to Core RBAC plus MRA in our proposed framework. $RBAC_1$ adds general role hierarchies to $RBAC_0$, and $RBAC_2$ enhances $RBAC_0$ by adding constraints such as mutually exclusive roles, cardinality and prerequisite roles. $RBAC_3$ combines all the features of previous models.

The first proposal for a standard on RBAC appeared at the 2000 ACM Workshop on RBAC [33]. It is organized into four levels of increasing capabilities. Flat RBAC (level 1) is comparable to Core RBAC in the standard. Hierarchical RBAC (level 2) requires supporting role hierarchies. Constrained RBAC (level 3) adds both SMER and DMER constraints (they were called SSD and DSD constraints). Symmetric RBAC (level 4) adds a requirement that one can review the permissions and roles that are available to a user or a role. In [16], Jaeger and Tidswell published a short rebuttal to the first proposal. They argued that "other than the first level, these co-called levels are orthogonal extensions to the basic RBAC model". Later versions of the standard adopted this suggestion. They also argued that "the proposed model does not add any value to user or permission aggregation, and it only limits the expression of hierarchies and constraints". We feel that this comment is probably partially due to the fact that many issues were left as implementation decisions and the draft standard does not provide any guidelines, a problem that remains in the final standard. Finally, they argued that administrative features of RBAC should be included in the standard. On this, we agree with the designers of the standard that these features are still not mature enough to be included in the standard.

The second proposal for an RBAC standard appeared in ACM Transactions on Information and Systems Security (TISSEC) in 2001 [14]. The final version [2] was approved in February 2004 as the American National Standard ANSI INCITS 359-2004.

A lot of work has been done on the issue of role hierarchies. Sandhu [35] discussed the UI and PI semantics of role hierarchies (under different names), and showed that in some situations it is desirable to have two separate hierarchies and the UI hierarchy extends the PI hierarchy. Moffett [25,26] examined the relationship between the inheritance properties of role hierarchies and control principles such as separation of duties, delegation and supervision. Crampton [7] recently showed that non-standard inheritance semantics (e.g., permissions are inherited by junior roles, rather than by senior roles) can be used to implement Mandatory Access Control in RBAC. Administration of RBAC is about controlling who can update the various relations in an RBAC system. A number of approaches for the administration of RBAC have been proposed [9,10,29,36,37,38,40].

Separation of Duty (SoD) was introduced into the information security literature in Saltzer and Schroeder [32]. SoD constraints in the context of RBAC were discussed in [1,8,15,17,23,41]. Li et al. [24] discussed the differences between mutual exclusion constraints as mechanisms and SoD as policy objectives and studied verification and generation problems related to using SMER constraints to enforce SSoD policies. They also proposed the terminology SMER and DMER, which we adopt.

6 Conclusions

We have identified and discussed some of the major issues in the current version of the ANSI RBAC standard [2,14]. In particular, we have discussed how to maintain and update a role hierarchy and how the different interpretations of a role hierarchy interact with other features such as constraints and sessions. We present a new RBAC framework that is inspired by the ANSI RBAC standard and is free of the problems that we have uncovered in the standard. An INCITS cybersecurity technical committee is being formed to discuss revisions to the ANSI RBAC standard and a submission to the ISO. We see our work in this paper as a significant contribution to the standardization effort.

References

[1] Gail-Joon Ahn and Ravi S. Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security*, 3(4):207–226, November 2000.

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

[3] Roland Awischus. Role based access control with the security administration manager (SAM). In *Proceedings of the second ACM workshop on Role-based access control table of contents (RBAC '97)*, pages 61–68, 1997.

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

[5] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, March 1976.

[6] David D. Clark and David R. Wilson. A comparision of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, May 1987.

[7] Jason Crampton. On permissions, inheritance and role hierarchies. In *Proceedings of the Tenth ACM Conference on Computer and Communications Security (CCS-10)*, pages 27–31. ACM Press, October 2003.

[8] Jason Crampton. Specifying and enforcing constraints in role-based access control. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, pages 43–50, Como, Italy, June 2003.

[9] Jason Crampton and George 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.

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

[11] David F. Ferraiolo, R. Chandramouli, Gail-Joon Ahn, and Serban Gavrila. The role control center: Features and case studies. In *Proceedings of the eighth ACM Symposium on Access Control Models and Technologies*, June 2003.

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

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

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

[15] Virgil D. Gligor, Serban I. Gavrila, and David F. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 172–183, May 1998.

[16] Trent Jaeger and Jonathon E. Tidswell. Rebuttal to the NIST RBAC model proposal. In *The fifth ACM workshop on Role-based access control*, pages 65–66, 2000.

[17] Trent Jaeger and Jonathon E. Tidswell. Practical safety in flexible access control models. *ACM Transactions on Information and System Security*, 4(2):158–190, May 2001.

[18] Gunter Karjoth. The authorization model of Tivoli Policy Director. In *Proceedings of the 17th Annual Computer Security Applications Conference*, pages 319–328, December 2001.

[19] Gunter Karjoth. Access control with IBM Tivoli Access Manager. *ACM Transactions on Information and System Security*, 6(2):232–257, May 2003.

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

[21] Axel Kern, Martin Kuhlmann, Andreas Schaad, and Jonathan Moffett. Observations on the role life-cycle in the context of enterprise security management. In *Proceedings of the Seventh ACM symposium on Access control models and technologies (SACMAT 2002)*, pages 43–51, 2002.

[22] Axel Kern, Andreas Schaad, and Jonathan 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.

[23] D. Richard Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 23–30, November 1997.

[24] Ninghui Li, Ziad Bizri, and Mahesh V. Tripunitara. On mutually-exclusive roles and separation of duty. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS-11)*, pages 42–51. ACM Press, October 2004.

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

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

[27] Matunda Nyanchama and Sylvia Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security*, 2(1):3–33, February 1999.

[28] National Institute of Standards and Technology. The economic impact of role-based access control. Planning Report 02-1, March 2002. available at http://www.nist.gov/director/prog-ofc/report02-1.pdf.

[29] Sejong Oh and Ravi 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.

[30] Oracle Coperation. *Oracle Database: Security Guide*, December 2003. Available at www.oracle.com.

[31] Chandramouli Ramaswamy and Ravi Sandhu. Role based access control features in commercial database management systems. In *21st National Information Systems Security Conference*, Oct 1998.

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

[33] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: towards a unified standard. In *Prcoeedings of the Fifth ACM Workshop on Role-Based Access Control (RBAC 2000)*, pages 47–63, 2000.

[34] Ravi S. Sandhu. Rationale for the RBAC96 family of access control models. In *Proceedings of the First ACM Workshop on Role-Based Access Control*, 1996.

[35] Ravi S. Sandhu. Role activation hierarchies. In *Proceedings of the Third ACM Workshop on Role-Based Access Control (RBAC 1998)*, pages 33–40, October 1998.

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

[37] Ravi S. Sandhu, Venkata Bhamidipati, Edward Coyne, Srinivas Ganta, and Charles 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, November 1997.

[38] Ravi S. Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based aministration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, February 1999.

[39] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[40] Ravi S. Sandhu and Qamar Munawer. The ARBAC99 model for administration of roles. In *Proceedings of the 18th Annual Computer Security Applications Conference*, pages 229–238, December 1999.

[41] Tichard T. Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *Proceedings of The 10th Computer Security Foundations Workshop*, pages 183–194. IEEE Computer Society Press, June 1997.

[42] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., 1989.

[43] 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.

Appendix

A Identified Errors in the ANSI RBAC standard [2]

In this section, we provide a non-exhaustive list of the errors we have found in ANSI RBAC standard.

Location	Identified Error	Correction
Page 2	...senor roles ...	...senior roles ...
Page 3	Rather then ...	Rather than ...
Page 3	...within a database management system, operations might include insert, delete, append and update.	There is no "append" operation in a typical DBMS. "select" operation seems to be more appropriate here.
Page 4	e.g., files, directories, in an operating system	e.g., files and directories in an operating system
Page 5	5.2 Hierarchal RBAC	5.2 Hierarchical RBAC
Page 5	session_users(s:SESSIONS)	This function returns a user for a given session. As there exists a single user for a session, the function should be named session_user(s:SESSION) to avoid any confusion.
Page 6	authorized_permissions(r) = $\{p \;\; PRMS \mid r' \;\; r, (p,r') \;\; PA\}$	authorized_permissions(r) = $\{p \;\; PRMS \mid r \;\; r', (p,r') \;\; PA\}$
Page 7	...as well as well as permission ...	...as well as permission ...

Location	Identified Error	Correction
Page 7	Node r_1 is represented as an immediate descendent of r_2 by r_1 r_2, ...	Node r_2 is represented as an immediate descendent of r_1 by r_1 r_2, ...
Page 7	...such that r_1 r_3 r_2, where $r_1 = r_2$ and $r_2 = r_3$	...such that r_1 r_3 r_2, where $r_1 = r_3$ and $r_2 = r_3$
Page 7	r, r_1, r_2 ROLES, r r_1 r r_2 $r_1 = r_2$	r, r_1, r_2 ROLES, r r_1 r r_2 $r_1 = r_2$
Page 12 –	OBJS	OBS
Page 14	CreateSession(*user*, *session*) and DeleteSession(*user*, *session*)	The names for these two functions are parameterized while no other functions is. Also, the parameters for CreateSession fail to include an active role set.
Page 14	*user_sessions*	This relation is never defined in the Reference Model.
Page 15–16	AssignedUsers(role : Name; out result : 2^{USERS}) ... AssignedRoles(user : NAME; result : 2^{ROLES})	The use of "out" in the two function signatures are inconsistent, see also CheckAccess, RolePermissions, and so on.
Page 17	UserOperationsOnObject This function returns the set of operations a given user is permitted to perform on a given object, obtained either directly or through his/her assigned roles.	This description implies that permissions can be assigned to users directly, not through roles. However, this is inconsistent with the Reference Model. This is also inconsistent with the pseudo-code for the function, which checks only the permissions assigned through roles.
Page 19	...and AddActiveRole of 7.1.2.	...and AddActiveRole of 6.1.2.
Page 19	CreateSession(user, session)	A set of active roles must be included as a parameter.
Page 20	In RolePermission, *result* = {*q*:*ROLES*;*op*:*OPS*;*obj*:*OBJS* | (*role* *q*) ((*op*, *obj*) *role*) *PA* • (*op*, *obj*)}	*result* = {*q*:*ROLES*;*op*:*OPS*;*obj*:*OBS* | (*role* *q*) ((*op*, *obj*) *q*) *PA* • (*op*, *obj*)}
Page 21	In RoleOperationsOnObject, *result* = {*q*:*ROLES*;*op*:*OPS* | (*role* *q*) ((*op*, *obj*) *role*) *PA* • *op*}	*result* = {*q*:*ROLES*;*op*:*OPS* | (*role* *q*) ((*op*, *obj*) *q*) *PA* • *op*}
Page 23	In AddSsdRoleMember |subset| = n	|subset| = ssd_card(set_name)

B Terminology on binary relations and partial orders

When we say a relation R, we mean a binary relation over a certain non-empty set X, i.e., R $X \times X$. When x, y R, we also write $R(x, y)$; when x, y R, we also write $\neg R(x, y)$. We use to denote logical implication and to denote logical equivalence.

- A relation R is transitive if x y z($R(x, y)$ $R(y, z)$ $R(x, z)$).

- A relation R is reflexive if $x R(x,x)$.
- A relation R is irreflexive if $x(\neg R(x,x))$.
- A relation R is symmetrical if $x\ y(R(x,y)\quad R(y,x))$.
- A relation R is asymmetrical if $x\ y(R(x,y)\ \neg\ R(y,x))$.
- A relation R is anti-symmetrical if $x\ y(R(x,y)\ \ R(y,x)\quad x = y)$.
- A strict partial order is irreflexive, transitive, and asymmetrical.
- A partial order is reflexive, transitive, and anti-symmetric.
- A relation R is a strict total order if it is a strict partial order and $xy(x = y\quad (R(x,y)\ \ R(y,x)))$.
- A relation R is a total order if it is a partial order and $xy(R(x,y)\ \ R(y,x))$.
- A relation R has a cycle if there exists a finite sequence of distinct elements $x_1, x_2, \ldots, x_k$ such that $(k > 1)\quad (\ j\ \{1,2,\ldots,k-1\}\ (R(x_j, x_{j+1})))\quad R(x_k, x_1)$.
- A relation R is acyclic if it does not have any ***cycle***.
- The transitive closure of a relation R is the smallest relation R such that R R and R is transitive.
- The reflexive closure of a relation R is the is the smallest relation R such that R R and R is reflexive.

C Functional Specification

Although the functions are described using the Z formal description language [42] in the standard, we use slightly different notation to improve the readability. Note that the functions that are added or corrected are marked with labels, ***(Added)*** or ***(Corrected)***, respectively.

C.1 Core RBAC

An RBAC system implementing Core RBAC should support the following administrative functions and review functions.

Administrative Functions

- AddUser: U. This function creates a new RBAC user with a given username.

```
AddUser(u:U)
  if u / USERS
  then USERS = USERS { u}
```

- DeleteUser: U. This function removes a user given a username.

```
DeleteUser(u:U)
  if u USERS
  then UA = UA \ {(u,r) | r ROLES (u,r) UA}
       USERS = USERS \ {u}
```

- AddRole: R. This function creates a new role with a given role name.

```
AddRole(r:R)
  if r / ROLES
  then ROLES = ROLES { r}
```

- DeleteRole: R. This function removes a role given a role name.

DeleteRole(r : R)
if r ROLES
then UA = UA \ {(u,r) | u USERS (u,r) UA}
PA = PA \ {(r,p) | p PRMS (r,p) PA}
ROLES = ROLES \ {r}

- AddPermission: P. This function creates a new permission with a given permission name.

AddPermission(p : P) *(Added)*
if p / PRMS
then PRMS = PRMS { p}

- DeletePermission: P. This function removes a permission given a permission name.

DeletePermission(p : P) *(Added)*
if p PRMS
then PA = PA \ {(r,p) | r ROLES (r,p) PA}
PRMS = PRMS \ {p}

- AssignUser: U × R. This function assigns a given user to a given role.

AssignUser(u : U; r : R)
if u USERS r ROLES (u,r) / UA
then UA = UA { (u,r)}

- DeassignUser: U × R. This function removes a user assignment given a user name and a role name.

DeassignUser(u : U; r : R)
if u USERS r ROLES (u,r) UA
then UA = UA \ {(u,r)}

- GrantPermission: P × R. This function assigns a given permission to a given role.

GrantPermission(p : P; r : R)
if p PRMS r ROLES (p,r) / UA
then PA = PA { (p,r)}

- RevokePermission: P × R. This function removes a permission assignment given a permission name and a role name.

RevokePermission(p : U; r : R)
if p PRMS r ROLES (p,r) PA
then PA = PA \ {(p,r)}

Review Functions

- AssignedUserRoles: USERS 2^{ROLES}. This function returns a set of roles to which a given user is assigned.

AssignedUserRoles(u : U; result : 2^{ROLES})
if u USERS
then result = {r | r ROLES (u,r) UA}

- AssignedRoleUsers: ROLES 2^{USERS}. This function returns a set of users that are assigned to a given role.

AssignedRoleUsers(r : R; result : 2^{USERS})
if r ROLES
then retsult = {u | u USERS (u,r) UA}

- AssignedRolePermissions: ROLES 2^{USERS}. This function returns a set of permissions that are assigned to a given role.

AssignedRolePermissions(r : R; result : 2^{PRMS})
if r ROLES
then result = {p | p PRMS (p,r) PA}

- AssignedPermissionRoles: PRMS 2^{ROLES}. This function returns a set of roles to which a given permission is assigned.

  AssignedPermissionRoles(p : P; result : 2^{ROLES}) *(Added)*
  if p PRMS
  then result = {r | r ROLES (p,r) PA}

- AssignedUserPermissions: USERS 2^{PRMS}. This function returns a set of permissions for which a given user is authorized through her role assignments.

  AssignedUserPermission(u : U; result : 2^{PRMS})
  if u USERS
  then result = {p | r ROLES p PRMS (u,r) UA (p,r) PA}

- AssignedPermissionUsers: PRMS 2^{USERS}. This function returns a set of users that are authorized for a given permission through their role assignments.

  AssignedPermissionUsers(p : P; result : 2^{USERS}) *(Added)*
  if p PRMS
  then result = {u | r ROLES u USERS (p,r) PA (u,r) UA}

The following functions are added to provide the compatibility to Hierarchical RBAC.

- AuthorizedUserRoles: U 2^{ROLES}.

  AuthorizedUserRoles(u : U; result : 2^{ROLES}) *(Added)*
  result = AssignedUserRoles(u)

- AuthorizedRoleUsers: R 2^{USERS}.

  AuthorizedRoleUsers(r : R; result : 2^{USERS}) *(Added)*
  result = AssignedRoleUsers(r)

- AuthorizedRolePermissions: R 2^{PRMS}.

  AuthorizedRolePermissions(r : R; result : 2^{PRMS}) *(Added)*
  result = AssignedRolePermissions(r)

- AuthorizedPermissionRoles: P 2^{ROLES}.

  AuthorizedPermissionRoles(p : P; result : 2^{ROLES}) *(Added)*
  result = AssignedPermissionRoles(p)

- AuthorizedUserPermissions: U 2^{PRMS}.

  AuthorizedUserPermissions(u : U; result : 2^{PRMS}) *(Added)*
  result = AssignedUserPermissions(u)

- AuthorizedPermissionUsers: P 2^{USERS}.

  AuthorizedPermissionUsers(p : P; result : 2^{USERS}) *(Added)*
  result = AssignedPermissionUsers(p)

- AuthorizedRoleRoles: R 2^{ROLES}

  AuthorizedRoleRoles(r : R; result : 2^{ROLES}) *(Added)*
  result = {r}

C.2 Role Hierarchies

An RBAC system implementing Hierarchical RBAC should support the functions of Core RBAC and the following functions. Note that some functions in Core RBAC are redefined here. We adopt the notations,   and , to denote the immediate inheritance relationship and the partial order relationship, respectively.

Administrative Functions

- DeleteRole: R . This function removes a role given a role name. Note that when a role is removed, any inheritance relationship (both explicit and implicit) established by the role is also removed. For instance, suppose that RH contains r_1 r_2 r_3. Deleting r_2 from the sytem removes both r_1 r_2 and r_2 r_3 from RH, which means that an implicit relationship r_1 r_3 is removed as well. This approach is indeed powerful and may not be desirable in some cases. Another approach is to remove a role only if there is no immediate inheritance relationships established by the role. In this case, one should remove all the related inheritance relationships before removing a role.

  DeleteRole(r : R) *(Corrected)*
  if r ROLES
  then RH = RH \ ({(r q) | q ROLES (r q) RH}
  { (q r) | q ROLES (q r) RH})
  ROLES = ROLES \ {r}

- AddInheritance: R × R . This function creates an immediate inheritance relationship between two given roles.

  AddInheritance(r_{asc} : R, r_{dsc} : R)
  if r_{acs} ROLES r_{dsc} ROLES (r_{acs} r_{dsc}) / RH ¬ (r_{acs} r_{dsc})
  [Limited hierarchies only: r ROLES, ($r_a$cs r) / RH]
  then RH = RH { (r_{acs} rdsc)}

- DeleteInheritance: R × R . This function removes the immediate inheritance relationship between two given roles.

  DeleteInheritance(r_{asc} : R, r_{dsc} : R)
  if r_{acs} ROLES r_{dsc} ROLES (r_{acs} r_{dsc}) RH
  then RH = RH \ {(r_{asc} r_{dsc})}

Review Functions

- AuthorizedUserRoles: USERS 2^{ROLES}. This function returns a set of roles for which a given user is authorized.

  AuthorizedUserRoles(u : U; result : 2^{ROLES}) *(Corrected)*
  if u USERS
  then result = {r | r, r ROLES (u, r) UA (r r)}

- AuthorizedRoleUsers: ROLES 2^{USERS}. This function returns a set of users that are authorized for a given role.

  AuthorizedRoleUsers(r : R; result : 2^{USERS})
  if r ROLES
  then result = {u | u USERS r ROLES (r r) (u, r) UA}

- AuthorizedRolePermissions: ROLES 2^{PRMS}. This function returns a set of permissions that are authorized by a given role.

  AuthorizedRolePermissions(r : R; result : 2^{PRMS}) *(Corrected)*
  if r ROLES
  then result = {p | p PRMS r ROLES (r r) (p, r) PA}

- AuthorizedPermissionRoles: PRMS 2^{ROLES}. This function returns a set of roles that authorizes a given permission.

  AuthorizedPermissionRoles(p : P; result : 2^{ROLES}) *(Added)*
  if p PRMS
  then result = {r | r, r ROLES (p, r) PA (r r)}

- AuthorizedUserPermissions: USERS 2^{PRMS}. This function returns a set of permissions for which a given user is authorized through her role assignments and the existing role hierarchies.

AuthorizedUserPermissions(u : U; result : 2^{PRMS}) *(Corrected)*

if u USERS

then result = {p | p PRMS r, r ROLES (u, r) UA (r r) (p, r) PA}

- AuthorizedPermissionUsers: PRMS 2^{USERS}. This function returns a set of users that are given a given permission through her role assignments and the role hierarchies.

AuthorizedPermissionUsers(p : P; result : 2^{USERS}) *(Added)*

if p PRMS

then result = {u | u USERS r, r ROLES (p, r) PA (r r) (u, r) UA}

- AuthorizedRoleRoles: R 2^{ROLES}. This function returns a set of roles that are dominated by a given roles.

AuthorizedRoleRoles(r : R; result : 2^{ROLES}) *(Added)*

result = {r | r r}

C.3 Static Constraint (SMER)

An RBAC system implementing Static Constraint (SMER) should support the functions of Core RBAC and the following functions. Note that some functions in Core RBAC are redefined here.

Administrative Functions

- DeleteRole: R. This function removes a role given a role name. Note that when a role is removed, every SMER constraint that includes the role must be updated.

DeleteRole(r : R) *(Corrected)*

if r ROLES

then UA = UA \ {(u, r) | u USERS (u, r) UA}

PA = PA \ {(p, r) | p PRMS (p, r) PA}

(c, rs, t) SMER, rs = rs \ {r}

ROLES = ROLES \ {r}

- AssignUser: U × R. This function assigns a given user to a given role.

AssignUser(u : U, r : R)

if u USERS r ROLESwedge(u r) / UA

(c, rs, t) SMER, ss rs where |ss| = t,

(AuthorizedRoleUsers(r) au) =)

r ss au = (if r = r then {u} else)

then UA = UA { (u, r)}

- CreateSMER: C × 2^{ROLES} × N. This function creates a new SMER constraint with a given name, conflicting role set and cardinality.

CreateSMER(c : C, rs : 2^{ROLES}, t : N)

if c / ExistingSMERs() rs ROLES (t ≥ 2) (t ≤ |rs|)

ss rs where |ss| = t, AuthorizedRoleUsers(r) =

r ss

then SMER = SMER { (c, rs, t)}

- DeleteSMER: C. This function removes a SMER constraint given a SMER name.

DeleteSMER(c : C)

if c ExistingSMERs()

then SMER = SMER \ {(c, rs, t)}

- AddRoleToSMER: C × R. This function adds a given role to the conflicting role set of a given SMER constraint.

A ddR oleT oSM E R (c :C;r :R)

if c E xistingSM E R s() r R O L E S r / SM E R R oles(c)

sr (SM E R R oles(c) { r}) w here |sr| = SM E R C ardinality(c)),

A uthorizedR oleU sers(r) =

r sr

then SM ER = SM ER \ {(c,SM E R R oles(c),SM E R C ardinality(c)}

{ (c,(SM E R R oles(c) { r}),SM E R C ardinality(c)}

- D eleteR oleFrom SM ER : C × R . This function rem oves a role from the role set associated w ith a given SM ER constraint.

D eleteR oleF rom SM E R (c :C;r :R)

if c E xistingSM E R s() r R O L E S r SM E R R oles(c)

SM E R C ardinality(c) < |SM E R R oles(c)|

then SM ER = SM ER \ {(c,SM E R R oles(c),SM E R C ardinality(c)}

{ (c,(SM E R R oles(c) \ {r}),SM E R C ardinality(c)}

- SetC ardinalityO fSM ER : C × N . This function sets the cardinality of a given SM ER constraint w ith a given num ber.

SetC ardinalityO fSM E R (c :C;t :N)

if c E xistingSM E R s() (t ≥ 2) (t ≤ |SM E R R oles(c)|)

sr SM E R R oles(c) w here |sr| = t, A uthorizedR oleU sers(r) =

r sr

then SM ER = SM ER \ {(c,SM E R R oles(c),SM E R C ardinality(c)}

{ (c,SM E R R oles(c),t}

- A ddInheritance : R × R . This function is specifically for system s w ith role hierarchies. This function establishes an im m ediate inheritance relationship betw een the tw o given roles.

A ddInheritance(r_{asc} :R ;r_{dsc} :R)

if r_{acs} R O L E S r_{dsc} R O L E S (r_{acs} r_{dsc}) / RH ¬ (r_{acs} r_{dsc})

(c,rs,t) SM ER , srs rs w here |sr| = t,

(A uthorizedR oleU sers(r) au) =)

r srs au= (if r= r_{dsc} then A uthorizedR oleU sers(r_{dsc}) else)

then RH = RH { r_{acs} rdsc}

R eview Functions

- ExistingSM ERs: 2^C . This function returns the nam es of all SM ER constraints in the system .

E xistingSM E R s(result : 2^C)

result = {c | (c,rs,t) SM ER }

- SM ERRoles: C 2^{ROLES} . This function returns the conflicting role set of a given SM ER constraint.

SM E R R oles(c :C;result : 2^{ROLES})

if c E xistingSM E R s()

then result = (rs | (c,rs,t) SM ER)

- SM ERCardinality: C N . This function returns the cardinality of a given SM ER constraint.

SM E R C ardinality(c :C;result :N)

if c E xistingSM E R s()

then result = (t | (c,rs,t) SM ER)

C.4 Session

An RBAC system implementing sessions should support the functions of Core RBAC and the following functions. Note that some functions in Core RBAC are redefined here.

Administrative Functions

- DeleteUser: U. This function removes a user given a user name. Note that when a user is removed, all the sessions belonging to the user are also removed.

```
DeleteUser(u : U)
   if   u   USERS
   then   s   UserSessions(u), DeleteSession(u, s)
        UA = UA \ {(u, r) | r   ROLES   (u, r)   UA}
        USERS = USERS \ {u}
```

- DeleteRole: R. This function removes a role given a role name. In the Functional Specification of the standard, all the affected sessions (i.e., the sessions whose session roles include the given role) are terminated, which seems extreme. Here we decide to allow the affected sessions to continue after the given role is removed.

```
DeleteRole(r : R)
   if   r   ROLES
   then   s   ExistingSessions(), DropActiveRole(s, r)
        UA = UA \ {(u, r) | u   USERS   (u, r)   UA}
        PA = PA \ {(p, r) | p   PRMS   (p, r)   PA}
        ROLES = ROLES \ {r}
```

- DeassignUser: U × R. This function removes a user assignment given a user name and a role name. Note that when a user is deassigned from a role, the role is removed from all the session roles of the user.

```
DeassignUser(u : U; r : R)
   if   u   USERS   r   ROLES   (u, r)   UA
   then   s   UserSessions(u), DropActiveRole(s, r)
        UA = UA \ {(u, r)}
```

- CreateSession: U × S × 2^{ROLES}. This function creates a new session for a given user with a given active role set. We assume AI and MRA sessions in that when a role is activated, all the junior roles of the role are also activated.

```
CreateSession(u : U; s : S; rs : 2^R)
   if   u   USERS   s / ExistingSessions()   rs   AuthorizedUserRoles(u)
   then SESSIONS = SESSIONS  {  (s, u,  )}
          r   rs, AddActiveRole(s, r)
```

- DeleteSession: U × S. This function removes an existing session of a given user.

```
DeleteSession(u : U; s : S)
   if   u   USERS   s   ExistingSessions()   u == SessionUser(s)
   then SESSIONS = SESSIONS \ {(s, u, SessionRoles(s))}
```

- AddActiveRole: U × S × R. This function adds a given role to the session role of a given user. We assume AI and MRA sessions in that when a role is activated, all the junior roles of the role are also activated.

(Corrected)

```
AddActiveRole(u : U; s : S; r : R)
   if   u   USERS   s   ExistingSessions()   r   ROLES   u == SessionUser(s)
        r   AuthorizedUserRoles(u)   r / SessionRoles(s)
   then SESSIONS = SESSIONS  \ {(s, u, SessionRoles(s))}
                 {  (s, u, (SessionRoles(s)    AuthorizedRoleRoles(r)))}
```

- DropActiveRole: U × S × R. This function removes a given role from the session role of a given user. We assume AI and MRA sessions in that when a role is deactivated, all the junior roles of the role are also deactivated.

DropActiveRole(u : U; s : S; r : R) *(Corrected)*

    if u USERS s SESSIONS r ROLES
        u == SessionUser(s); r SessionRoles(s)
    then SESSIONS = SESSIONS \ {(s, u, SessionRoles(s))}
            { (s, u, (SessionRoles(s) \ AuthorizedRoleRoles(r)))}

Review Functions

- ExistingSessions: 2^S. This function returns the names of all currently existing sessions in the system.

    ExistingSessions(result : 2^S)
        result = {s | (s, u, rs) SESSIONS}

- SessionRoles: S 2^{ROLES}. This function returns a set of roles that are activated in a given session.

    SessionRoles(s : S; result : 2^{ROLES})
        if s SESSIONS
        then result = (rs | (s, u, rs) SESSIONS)

- SessionPermissions: S 2^{PRMS}. This function returns a set of permissions that are available in a given session.

    SessionPermissions(s : S; result : 2^{PRMS})
        if s ExistingSessions()
        then result = {p | r SessionRoles(s)
                p AuthorizedRolePermissions(r)}

- SessionUser: S U. This function returns the user (i.e., owner) of a given session.

    SessionUser(s : S; result : U) *(Added)*
        if ss SESSIONS
        then result = (u | u USERS (s, u, rs) SESSIONS)

- UserSessions: U $2^{SESSIONS}$. This function returns all the sessions that belong to a given user.

    UserSessions(u : U; result : $2^{SESSIONS}$) *(Added)*
        if u USERS
        then result = {s | (s, u, rs) SESSIONS}

C.5 Dynamic Constraint (DMER)

An RBAC system implementing Dynamic Constraint (DMER) should support the functions of Core RBAC as well as MRA sessions and the following functions. Note that some functions in Core RBAC and Session are redefined here.

Administrative Functions

- DeleteRole: R. This function removes a role given a role name. Note that when a role is removed, every DMER constraint that includes the role must be updated.

    DeleteRole(r : R) *(Corrected)*
        if r ROLES
        then UA = UA \ {(u, r) | u USERS (u, r) UA}
            PA = PA \ {(p, r) | p PRMS (p, r) PA}
             (d, rs, t) DMER, rs = rs \ {r}
            ROLES = ROLES \ {r}

- AddActiveRole: U × S × R. This function adds a given role to the session role of a given user.

AddActiveRole(u:U;s:S;r:R)
  if u USERS s SESSIONS r ROLES u == SessionUser(s)
    r AuthorizedUserRoles(u) r / SessionRoles(s)
    (d,rs,t) DMER, dr rs,
      dr (SessionRoles(s) { r}) | dr| < t
SESSIONS = SESSIONS \ {(s,u,SessionRoles(s))}
      { (s,u,(SessionRoles(s) AuthorizedRoleRoles(r)))}

- CreateDMER: D × 2^{ROLES} × N. This function creates a new DMER constraint with a given name, conflicting role set and cardinality.

CreateDMER(d:D;rs:2^{ROLES};t:N)
  if d / ExistingDMERs() rs ROLES (t ≥ 2) (t ≤ |rs|)
    s SESSIONS, dr rs, (dr SessionRoles(s)) | dr| < t
  then DMER = DMER { (d,rs,t)}

- DeleteDMER: D. This function removes a DMER constraint given a DMER name.

DeleteDMER(d:D)
  if d ExistingDMERs()
  then DMER = DMER \ {(d,rs,t)}

- AddRoleToDMER: D × R. This function adds a given role to the conflicting role set of a given DMER constraint.

AddRoleToDMER(d:D;r:R)
  if d ExistingDMERs() r ROLES r / DMERRoles(d)
    s SESSIONS, dr (DMERRoles(d) { r}),
      (dr SessionRoles(s)) | dr| < t)
  then DMER = DMER \ {(d,DMERRoles(d),DMERCardinality(d)}
      { (d,(DMERRoles(d) { r}),DMERCardinality(d)}

- DeleteRoleFromDMER: D × R. This function removes a role from the conflicting role set of a given DMER constraint.

DeleteRoleFromDMER(d:D;r:R)
  if d ExistingDMERs() r ROLES
    r DMERRoles(d) DMERCardinality(d) < |DMERRoles(d)|
  then DMER = DMER \ {(d,DMERRoles(d),DMERCardinality(d)}
      { (d,(DMERRoles(d) \ {r}),DMERCardinality(d)}

- SetCardinalityOfDMER: D × N. This function sets the cardinality of a given DMER constraint with a given number.

SetCardinalityOfDMER(d:D;t:N)
  if d ExistingDMERs() (t ≥ 2) (t ≤ |DMERRoles(d)|)
    s SESSIONS, dr DMERRoles(d),
      (dr SessionRoles(s) | dr| < t)
  then DMER = DMER \ {(d,DMERRoles(d),DMERCardinality(d)}
      { (d,DMERRoles(d),t}

Review Functions

- ExistingDMERs: 2^{D}. This function returns the names of all DMER constraints in the system.

ExistingDMERs(result:2^{D})
    result = {d | (d,rs,t) DMER}

- DMERRoles: D 2^{ROLES}. This function returns the role set associated with a given DMER constraint.

DMERRoles(d : D; result : 2^{ROLES})
if d ExistingDMERs()
then result = (rs | (d,rs,t) DMER)

- DMERCardinality: D N. This function returns the cardinality of a given DMER constraint.

DMERCardinality(d : D; result : N)
if d ExistingDMERs()
then result = (t | (d,rs,t) DMER)