Role-Based Access Control and the Access Control Matrix

G. Saunders
Basser Department of Computer Science, University of Sydney, Australia
gsaunder@cs.usyd.edu.au

M. Hitchens and V. Varadharajan
School of Computing & Information Technology, University of Western Sydney (Nepean), Australia
{m.hitchens,v.varadharajan}@uws.edu.au

Abstract

The Access Matrix is a useful model for understanding the behaviour and properties of access control systems. While the matrix is rarely implemented, access control in real systems is usually based on access control mechanisms, such as access control lists or capabilities, that have clear relationships with the matrix model. In recent times a great deal of interest has been shown in Role Based Access Control (RBAC) models. However, the relationship between RBAC models and the Access Matrix is not clear. In this paper we present a model of RBAC based on the Access Matrix which makes the relationships between the two explicit. In the process of constructing this model, some fundamental similarities between certain capability models and RBAC are revealed.

1 Introduction

Computer systems contain large amounts of information. Much of this information is of a sensitive nature. For such information it is necessary to be able to define what entities have access to the information and in what ways they can access the information. These functions are variously known as access control or authorisation. The basic model of access control is the Access Control Matrix (or ACM) [9, 6]. The access control matrix specifies individual relationships between entities wishing access (subjects) and the system resources they wish to access (objects). For each subject-object pair the allowable access appears in the corresponding entry in the (two-dimensional) matrix. Current access control mechanisms do not implement the access control matrix directly, due to well known efficiency problems [13]. However, most access control mechanisms in current use are based on models, such as access control lists or capabilities [3, 4], which have a direct relationship with the access control matrix.

Recently there has been an increasing interest in other models of access control. One of the more prominent of these has been Role Based Access Control (or RBAC) [5, 11, 12]. The interest in these alternative approaches to access control has, at least in part, arisen due to the limited expressive power of the common access control models. The policies of real world organisations are often of a sophisticated nature and cannot be readily expressed within the simple framework of the access control matrix or its immediate derivatives. RBAC, amongst other proposals [7], shows promise in being able to express real-world policies. The basic elements of an RBAC system are

1. Users
2. Roles
3. Permissions

Permissions specify the access allowed to objects and are grouped together in roles, which can be thought of as job descriptions within an organisational structure. Users are assigned to roles. This approach has a number of immediate benefits. Roles can be given permissions which match the organisational policy for the equivalent job. Users can be switched easily between roles, automatically changing the permissions available to them. This is much easier than traditional access control systems, where the individual permissions for that user would each have to be updated to reflect the abilities of the new position.

As might be expected, the advantages of RBAC do not come without cost. The ACM is a relatively simple concept and it, and its closely related derivatives (access control lists and capabilities) have been extensively studied. Even here though, the differences between access control lists and capabilities have made it difficult to compare systems based on these models in any formal way. In a previous paper [14] we presented a formalism, based on that of Harrison, Ruzzo and Ullman [6] which encompasses both access control lists and capabilities, making it easier to compare such systems.

In this paper we extend that formalism to encompass Role Based Access Control. In the process it becomes clear that role based access control has significant fundamental similarities to capability based access control.

We begin in the following section with a basic model from which the other models are derived. This basic model is revision and simplification of a model we presented in an earlier work [14] which in turn was based on the matrix model of Harrison et al. [6].

Section 3 extends the basic model to form a matrix model and suggests some extensions that reduce the space requirements of the model. From the resulting matrix model, an Access Control List (ACL) model and a basic capability model are derived in Sections 4 and 5. In Section 6 a more complex capability model is derived which has some fundamental properties of RBAC models, and in Section 7 we derive an RBAC model with notable similarities to this capability model. Section 8 presents some examples which show these similarities more clearly. Section 9 completes the paper with some concluding remarks and areas for further research.

2 The Base Model

We begin with a basic model which is expanded in later sections to describe the various access control models. The model presented here is a revision and simplification of one we presented in an earlier work [14], which in turn was based on the access matrix model of Harrison et al. [6].

We shall base our Access Control Models on a series of definitions, each of which declares the existence of one of three things:

1. A set

2. A container (list, queue, vector, matrix etc.) the contents of which are either
    - Elements of a set defined earlier; or
    - A set or container (recursively.)
3. A mapping between sets defined earlier.

Each of the models in this formulation are extensions of the following six definitions, some of which may be augmented or redefined depending on the model in question.

Definition 2.1 Rts *the set of Rights (e.g. read, write, execute)*

Definition 2.2 Obj *the set of Objects (e.g. files)*

Definition 2.3 Sbj *the set of Subjects (e.g. users, processes)*

Definition 2.4 C *the set of commands*

Definition 2.5 B *the set* $\{grant, deny\}$

Definition 2.6 f *a function from* $\mathrm{Sbj} \times \mathrm{Obj} \times \mathrm{Rts}$ *to* B

Each element of C is a command of the form:

```
command $\alpha(X_1, \ldots, X_i)$
    if $cnd_1$ and
        $\ldots$
        $cnd_j$
    then
        $op_1 \ldots op_k$
end
```

The commands provide the only means of manipulating the elements of the access control system in the same way that the methods of an object oriented class provide the only means for manipulating the private variables of that class. The contents of C are determined by the model under consideration, and each model will typically provide commands for creating and destroying objects and subjects, and for conferring and revoking access privileges between subjects.

`enter` x `into` Y	`delete` x `from` Y
`create object` X_o	`destroy object` X_o
`create subject` X_s	`destroy subject` X_s

Table 1: The *primitive operations* available to the commands in C

The symbol α is the name of the command. The arguments, $X_1 \ldots X_i$, may be elements of any set declared earlier. Within the commands, each cnd_j is a condition using either the function f or the operator 'in' which is used to test membership in a set or container. Each op_k is one of the primitive operations in Table 1. The `enter` and `delete` operations are defined more generally than in the model of Harrison et al. They allow an element x to be entered or removed from a set or container Y. We assume that the other operations have intuitive meanings.

The function f is used to determine whether a given subject has a given right for a given object. The exact definition of this function depends on the model in question.

3 The Access Control Matrix Model

To model the Access Control Matrix [6] we begin with Definitions 2.1–2.6 from the previous section, and extend them with

Definition 3.1 *M a matrix, indexed by* Obj *and* Sbj*, each element of which is a subset of* Rts*.*

The contents of C from Definition 2.4 are shown in Table 2. Commands for the creation and destruction of subjects are similar to those for objects and are omitted here and in the later models. The function f from 2.6 takes the form:

$$f(s,o,rt) = \begin{cases} grant & \text{if } rt \in M[s,o] \\ deny & \text{otherwise} \end{cases}$$

So much for the Access Control Matrix model then. As is well known, the space requirements of the matrix prohibit the actual use of this model in a computer system. There are, however, methods for reducing the space required. For example, we can replace Definition 3.1 with:

Definition 3.2 (replaces 3.1) *M a set of triplets (s, o, rts) where $s \in \mathrm{Sbj}$, $o \in \mathrm{Obj}$ and $\mathrm{rts} \subseteq \mathrm{Rts}$.*

and then remove those triplets where $rts = \emptyset$ to save space, based on the assumption that the majority of entries in the matrix are, in fact, empty [13]. This would require modifications to the commands in C, for example in the `CREATE` command we add:

```
enter $(sbj, obj, \{own\})$ into $M$
```

in place of the existing `enter` operation. However, we can do even better than this. Suppose we group subjects together, with:

Definition 3.3 *G a set of groups, where each group is a subset of* Sbj*.*

```
command $CREATE(sbj, obj)$
  create object $obj$
  enter $own$ into $M[sbj, obj]$
end

command $DESTROY(sbj, obj)$
  if $own$ in $M[sbj, obj]$ then
    destroy object $obj$
end

command $CONFER_r(sbj, sbj2, obj)$
  if $own$ in $M[sbj, obj]$ then
    enter $r$ into $M[sbj2, obj]$
end

command $REMOVE_r(sbj, sbj2, obj)$
  if $own$ in $M[sbj, obj]$ then
    delete $r$ from $M[sbj2, obj]$
end

command $CHOWN(sbj, new, obj)$
  if $own$ in $M[sbj, obj]$ then
      delete $own$ from $M[sbj, obj]$
      enter $own$ into $M[new, obj]$
end
```

Table 2: The set C of commands for the access control matrix model.

and modify the commands in C so that a group may be given in place of a subject in the parameter list.[1] Then we replace Definition 3.2 with

Definition 3.4 (replaces 3.2) *M a set of triplets (x, o, rts) where $x \in \mathrm{Sbj} \cup G$.*

[1]If groups are not to be owners of objects, they must not be allowed to replace subjects in the `CREATE` and `CHOWN` commands.

By using groups in this way, we reduce the number of triplets in M, resulting in a model that does not have the prohibitive space requirements of the access control matrix. Unfortunately it is likely to have prohibitive computational time requirements (which is to say that the execution of f will take a considerable amount of time). We turn now to methods for reducing these time requirements.

4 The Access Control List Model

In order to reduce the computational time requirements of the revised matrix model, we must reduce the number of entries in M that are considered during the execution of f. We can do this by partitioning M into small discrete portions based on either subject or object. In this section we consider partitioning M by object.

We begin again with Definitions 2.1–2.6, and extend them with:

Definition 4.1 Cols *a vector indexed by* Obj*. Each entry in* Cols *is a vector indexed by the elements of* Sbj*. The entries of each vector are subsets of* Rts*.*

This is just the matrix split into column vectors. We can remove the empty elements from these column vectors to obtain:

Definition 4.2 (replaces 4.1) Cols *a vector indexed by* Obj*. Each element of* Cols *is a set of* (s, rts) *tuples where* $s \in \mathrm{Sbj}$ *and* $\mathrm{rts} \in \mathrm{Rts}$*.*

To further reduce space requirements we can use groups from Definition 3.3 above to obtain:

```
command $CREATE(sbj, obj)$
  create object $obj$
  enter $(sbj, own)$ into $Cols[obj]$
end

command $DESTROY(sbj, obj)$
  if $f(sbj, obj, own)$ then
    destroy object $obj$
end

command $CONFER_r(sbj, sbj2, obj)$
  if $f(sbj, obj, own)$ then
    enter $sbj2, \{r\}$ into $Cols[obj]$
end

command $REMOVE_r(sbj, sbj2, obj)$
  if $f(sbj, obj, own)$ then
    delete $sbj2, \{r\}$ from $Cols[obj]$
end

command $CHOWN(sbj, new, obj)$
  if $f(sbj, obj, own)$ then
   delete $(sbj, \{own\})$ from $Cols[obj]$
   enter $(new, \{own\})$ into $Cols[obj]$
end
```

Table 3: The set C of commands for the access control list model.

Definition 4.3 (replaces 4.2) Cols *a vector indexed by* Obj*. Each element of* Cols *is a set of* (x, rts) *tuples where* $x \in \mathrm{Sbj} \cup G$ *and* $\mathrm{rts} \in \mathrm{Rts}$*.*

The set of commands, C, is shown in Table 3 and the function f takes the form:

$$f(s, o, rt) = \begin{cases} grant & \text{if} rt \in \{y | (x, rts) \in \\ & Cols[o] \wedge (s = x \in Sbj \\ & \vee s \in x \in G) \wedge y \in rts\} \\ deny & \text{otherwise} \end{cases}$$

and thus we arrive at the basic Access Control List (ACL) model used in systems such as Unix. However, it is possible to reduce the storage requirements even further by storing only the unique (x, rts) tuples. This step is rarely taken in ACL systems, however a similar approach is used to reduce storage requirements in capability systems. Appropriate definitions for this are:

Definition 4.4 SR *a set of* (s, rts) *tuples formed by taking the union of all sets in* Cols*.*

Definition 4.5 SRA *a many-to-many mapping from* SR *to* Obj*.*

Access Control Lists view the Access Matrix in a column-wise fashion. An alternative approach would be to view them in a row-wise manner. This is the approach taken by capabilities.

5 A Capability Model

Capability systems (e.g. [3, 4]) partition M by subject rather than object. We can do this in a similar manner to the one used above for the ACL model. We begin with definitions 2.1–2.6 and extend them with:

Definition 5.1 Rows *a vector indexed by the elements of* Sbj*. Each entry of* Rows *is a vector indexed by the elements of* Obj*. The entries of each vector are subsets of* Rts*.*

and, as with the list model described above, we remove empty elements to obtain:

Definition 5.2 (replaces 5.1) Rows *a vector indexed by* Sbj*. Each element of* Rows *is a set of* (o, rts) *tuples where* $o \in \mathrm{Obj}$ *and* $\mathrm{rts} \subseteq \mathrm{Rts}$*.*

```
command $CREATE(sbj, obj)$
 create object $obj$
 enter $(obj, \{Rts\})$ into $OR$
 enter $((obj, \{Rts\}), sbj)$ into $ORA$
end

command $DESTROY(sbj, obj)$
 if $f(sbj, obj, destroy)$ then
   destroy object $obj$
end

command $CONFER_r(sbj, sbj2, obj)$
 if $f(sbj, obj, confer)$ then
  enter $(obj, \{r\})$ into $OR$
  enter $((obj, \{r\}), sbj2)$ into $ORA$
end

command $REMOVE_r(sbj, sbj2, obj)$
 if $f(sbj, obj, remove)$ then
  delete $((obj, \{r\}), sbj2)$ from $ORA$
end
```

Table 4: The set C of commands for the capability model.

Storing only the unique (o, rts) tuples, is common in capability systems as it reduces storage requirements. This approach can be modelled by means of a central capability table, using:

Definition 5.3 OR *a set of* (o, rts) *tuples, formed by taking the union of all the sets in* Rows*.*

Definition 5.4 ORA *a many-to-many mapping from* OR *to* Sbj*.*

The set of commands, C is defined in Table 4 and the function f takes the form:

$$f(s, o, rt) = \begin{cases} grant & \text{if} \, rt \in \{x | (o, rts \in OR \wedge \\ & ((o, rts), s) \in ORA \wedge \\ & x \in rts\} \\ deny & \text{otherwise} \end{cases}$$

This gives us a model that behaves in a similar manner to early capability systems which utilised a central capability table. Unfortunately, central capability systems have suffered from performance problems [8], and more recent capability systems take a different approach which is described in the following section.

The groups concept can be applied in multiple ways with capabilities. One interesting way is to have groups of objects instead of subjects. An object in one of the commands in Table 4 could be replaced by a group of objects.

6 Capability Container Systems

Some capability systems, (e.g. [2] and [1]), allow capabilities to be stored within objects. When a subject wishes to access an object, they locate a capability within one of their objects and present it to the system. Beginning again with Definitions 2.1–2.6 and Definition 5.3, this can be modelled in the following way:

Definition 6.1 CapO *a set of objects that may contain capabilities, and such that* CapO $\subseteq O$.

Definition 6.2 CA *a many-to-many mapping from* OR *to* CapO.

Instead of storing capabilities in a central repository (OR), they are stored within objects. The CA mapping simply tells us which capabilities are contained in a particular $CapO$.

This scheme raises a number of interesting issues. Firstly, what capabilities does a subject possess on creation? One possibility would be to create a mapping from some characteristic of the new subject, its owner for example, to a set of capabilities which the subject will possess on creation. Another solution would have the subject inherit some or all of the capabilities of its parent. These solutions are not mutually exclusive, and the second has the advantage of being able to support the principle of *least privilege* by dynamically restricting the capabilities that a child subject inherits, or perhaps by temporarily deactivating capabilities under certain conditions (for example, the password capability system of Anderson et al. [1] has facilities for doing this). To model this we introduce:

Definition 6.3 proclist *a many-to-many mapping from* Sbj *to* OR*.*

which tells us which of the capabilities held in a subject can be used at the present time.[2]

Another issue raised by this scheme is that by possessing a capability for a capability containing object a subject may, depending on the rights in the capability, be able to acquire and use the capabilities in that object. We designate the set of capability containing objects from which a subject can acquire capabilities as the *active* capability containing objects.

Definition 6.4 active *a many-to-many mapping from* Sbj *to* CapO *giving the* CapO*s which are* reachable *from a given subject. A* CapO *called x is reachable by a subject s if $x = s$, or if a capability for x is an element of* proclist(s),

[2] *proclist* is implementation dependent, so the mechanism by which it determines its results is not discussed here.

```
command $CREATE(sbj, obj)$
  create object $obj$
  enter $(obj, \{Rts\})$ into $sbj$
end

command $DESTROY(sbj, obj)$
  if $f(sbj, obj, destroy)$ then
    destroy object $obj$
end

command $CONFER_r(sbj, capo, obj)$
  if $f(sbj, obj, confer)$ then
    enter $(obj, \{r\})$ into $capo$
end

command $REMOVE_r(sbj, sbj2, obj)$
  if $f(sbj, obj, remove)$ then
    delete $(obj, \{r\})$ from $capo$
end
```

Table 5: The set C of commands for the capability container model.

or if a capability for x is an element of $\mathrm{CA}(y)$ *(where $y \in \mathrm{CapO}$) and y is reachable.*

A process wishing to access an object would simply present a capability for the object from among the capabilities available in *any* of the objects to which it has a capability, or can get one. The function f therefore takes the form

$$f(s, o, rt) = \begin{cases} grant & \text{if } (o, rts) \in proclist(s) \vee \\ & (o, rts) \in \cup_{c \in active(s)} \{x | \\ & x \in CA(c)\} \wedge rt \in rts \\ deny & \text{otherwise} \end{cases}$$

and the set C of commands is shown in Table 5.

7 Role Based Access Control Models

We are now ready to discuss Role Based Access Control (RBAC) models.

Sandhu et al. [11] define four reference models for Role Based Access Control. RBAC_0 defines a basic RBAC system. RBAC_1 augments RBAC_0 with role hierarchies. RBAC_2 adds constraints to RBAC_0 and RBAC_3 combines RBAC_1 and RBAC_2. In this paper we focus on RBAC_1, which has the following components [11]

- U, R, P and S (users, roles, permissions, and sessions respectively);
- $PA \subseteq P \times R$, a many-to-many permission to role assignment relation;
- $UA \subseteq U \times R$, a many-to-many user to role assignment relation;
- $user : S \rightarrow U$, a function mapping each session s_i to the single user $user(s_i)$ (constant for the session's lifetime);
- $RH \subseteq R \times R$ is a partial order on R called the role hierarchy or role dominance relation, also written as $\geq$; and
- $roles : S \rightarrow 2^R$, a function mapping each session s_i to a set of roles $roles(s_i) \subseteq \{r | (\exists r' \geq r)[(user(s_i), r') \in UA]\}$ (which can change with time) and session s_i has the permissions $\cup_{r \in roles(si)} \{p | (\exists r'' \geq r)[(p, r'') \in PA]\}$.

It may come as a surprise to realize that there are fundamental similarities between the Container Based Capability model presented earlier, and Role Based Access Control Models. In fact, the process of deriving a Role Based Access Control Model from the model presented in the previous section is largely one of renaming.

We extend our previous definitions with:

Definition 7.1 (replaces 5.3) *P the set of Permissions, such that $P = \mathrm{OR}$.*

The subjects in a Role Based Access Control system are neither users, nor processes, but a new entity called a session. When a user logs in, they create a new session which is active in a subset of their roles (see below). This is analogous to a user logging in and creating a process with a subset of their capabilities. The function f must be redefined in light of this:

Definition 7.2 *S the set of Sessions.*

Definition 7.3 (replaces 2.6) *f a function from $S \times \mathrm{Obj} \times \mathrm{Rts}$ to B*

In RBAC systems, roles relate users to permissions. This is analogous to giving a user a capability for a capability containing object.

Definition 7.4 *R a set of Roles.*

Perhaps the most important difference between container based capability models and RBAC models is that roles are not objects as *CapO*s are. Therefore, it is not possible to manipulate roles in the same way that normal system objects can be manipulated.

Also, a permission is not required to access a Role. Instead, role membership is determined independently of any permissions held by a user (indeed, the permissions held by a user are determined by role membership.)

Lastly, roles are not, strictly speaking, sets of permissions (though they can be usefully thought of as such). So we require a mechanism to tell us which permissions are assigned to a role, just as we required a mechanism to map capabilities to the objects which contained them.

```
command $CREATE(role, obj)$
 create object $obj$
 enter $(obj, \{Rts\})$ into $P$
 enter $((obj, \{Rts\}), role)$ into $PA$
end

command $DESTROY(role, obj)$
 if $f(role, obj, destroy)$ then
  destroy object $obj$
end

command $CONFER_r(role1, obj, role2)$
 if $f(role1, obj, confer)$ then
  enter $((obj, \{r\}), role2)$ into $PA$
end

command $REMOVE_r(role1, obj, role2)$
 if $f(role1, obj, remove)$ then
  delete $((obj, \{r\}), role2)$ from $PA$
end
```

Table 6: The set C of commands for the role based model.

Definition 7.5 (replaces 6.2) PA *a many-to-many mapping from P to R.*

In the container based capability models, the subjects are themselves capability containers and therefore behave in a similar manner to roles. In RBAC they are restricted to merely inheriting permissions from roles, they are not able to contain permissions that are not inherited from roles. Furthermore, it is not possible to obtain permissions from a Subject by possessing a permission for that subject.

We require a mechanism to tell us which roles are being used by a particular session. This mechanism performs a similar function to *proclist* from Definition 6.3, in that it allows for a subset of the available roles to be made active.

Definition 7.6 roles *a many-to-many mapping from S to R.*

Some RBAC models allow roles to be partially ordered in a Role Hierarchy. This is analogous to having a capability containing object which contains capabilities for other capability containing objects. The *active* mapping from Definition 6.4 provides an almost identical function in container based capability models. In RBAC models the role hierarchy is defined by

Definition 7.7 RH *A partial order on the set R of roles.*

The commands of the set C are defined in Table 6 and the function f from Definition 7.3 takes the form

$$f(s,o,rt) = \begin{cases} grant & \text{if } (o,rts) \in \cup_{rl \in roles(s)} \\ & \{p \mid p \in PA[rl]\} \wedge \\ & rt \in rts \\ deny & \text{otherwise} \end{cases}$$

We now have a basic RBAC model derived from the container based capability model of the previous section. The following sections present examples to illustrate the similarities between RBAC and container based capability models.

8 Some Examples

In this section we illustrate the similarities between Role Based models and capability container models with examples taken from Sandhu et al.[11].

8.1 Chief Security Officer Example

Consider the role hierarchy found in Figure 1(a) in which a Chief Security Officer (CSO) role inherits from three junior Security Officer (SO) roles. In this example, the set R of roles is simply $\{CSO, SO1, SO2, SO3\}$, and the partial order set RH contains $\{(SO1, CSO), (SO2, CSO), (SO3, CSO)\}$.

Figure 1(b) is an example access control matrix describing the rights each of the security officer roles has for objects O_1, O_2 and O_3. In a Role Based system, this matrix is represented by the set P of permissions which contains

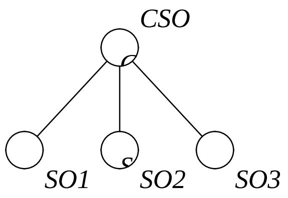
$$\left\{ \begin{array}{cc} \underbrace{O_1, \{read\}}_{p_1} & \underbrace{O_2, \{read, write\}}_{p_2} \\ \underbrace{O_2, \{read, execute\}}_{p_3} & \underbrace{O_3, \{read, write\}}_{p_4} \end{array} \right\}$$

and the permission assignment set, PA, contains

$$\left\{ \begin{array}{c} (p_1, SO1), (p_1, SO2), (p_2, CSO), \\ (p_3, SO2), (p_4, SO3) \end{array} \right\}$$

This means that any user active in the CSO role is able to use permission p_2 and also any of the other permissions by virtue of the inheritance relationships.

Figure 2 illustrates the same scenario in terms of the capability container model. The set of capability containing objects, $CapO$, would be $\{CSO, SO1, SO2, SO3\}$. The capabilities contained within the CSO object include $\{(SO1, acq), (SO2, acq), (SO3, acq)\}$ where acq represents the set of rights which enable the acquisition and use of capabilities from the destination object.



(a) Role Hierarchy

	O_1	O_2	O_3
CSO		read, write	
SO1	read		
SO2	read	read, execute	
SO3			read, write

(b) Access Matrix

Figure 1: The chief security officer example

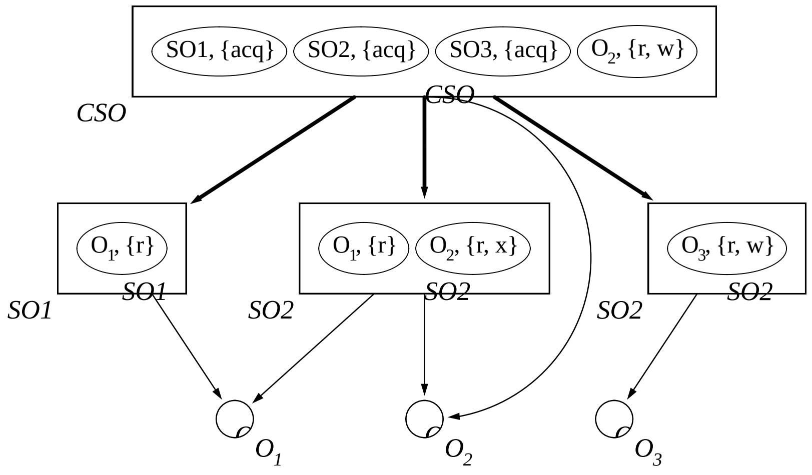


Figure 2: The example using the capability container model.

In addition to the capabilities mentioned above, the set of capabilities *OR* contains the permissions of set P. Furthermore, the *SO1* object contains the capability p_1, the *SO2* object contains the capabilities p_1 and p_3, the *SO3* object contains p_4 and lastly, the *CSO* object contains p_2. Since *CSO* also contains capabilities for *SO1*, *SO2* and *SO3*, any user who holds a capability for CSO is able to retrieve and use capabilities from *SO1*, *SO2* and *SO3* in an analogous way to role inheritance.

8.2 Project Supervisor Example

In this section we present a more complex example, again taken from Sandhu et al. [11]. Figure 3(a) is a role hierarchy in which a project supervisor role (S) inherits two task roles (T1 and T2) which, in turn, inherit a general project role (P). In addition, the supervisor role inherits a subproject supervisor role (S3), which inherits another two task roles (T3 and T4), which inherit from a subproject role (P3), which, finally, inherits the project role.

The set R of roles in this example is $\{S, S3, T1, T2, T3, T4, P3, P\}$, and the role hierarchy set RH is

$$\left\{ \begin{array}{c} (S3, S), (T1, S), (T2, S), (T3, S3), \\ (T4, S3), (P, T1), (P, T2), (P3, T3), \\ (P3, T4), (P, P3) \end{array} \right\}$$

The set P of permissions is

$$\left\{ \begin{array}{ccc} \underbrace{O_1, \{r\}}_{p_1} & \underbrace{O_1, \{r, w, x\}}_{p_2} & \underbrace{O_2, \{w, x\}}_{p_3} \\ \underbrace{O_2, \{r\}}_{p_4} & \underbrace{O_3, \{r, w\}}_{p_5} & \underbrace{O_4, \{x\}}_{p_6} \\ \underbrace{O_4, \{w, x\}}_{p_7} & \underbrace{O_4, \{r\}}_{p_8} & \end{array} \right\}$$

Lastly, the permission assignment relation, *PA*, contains

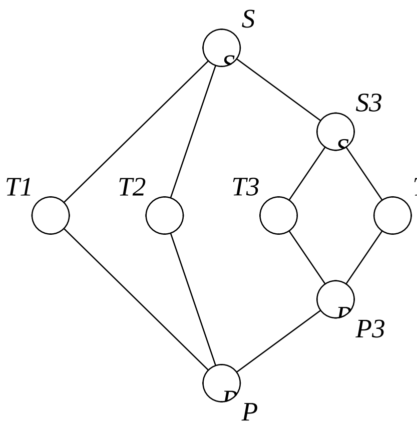
$$\left\{ \begin{array}{c} (p_1, T1), (p_2, T2), (p_3, T2), (p_4, P), \\ (p_5, T3), (p_6, T3), (p_7, T4), (p_8, P3) \end{array} \right\}$$

Figure 4 illustrates the capability container version of this example. There are eight capability containing objects, $\{S, S3, T1, T2, T3, T4, P3$ and $P\}$. Object S contains the capabilities for *T1*, *T2* and *S3*. Object *S3* contains capabilities for *T3* and *T4*. Objects *T1*, *T2* and *P3* each contain capabilities for P. Finally, objects *T3* and *T4* each contain a capability for *P3*. Each of these capabilities has the *acq* permission, which permits the acquisition and use of capabilities from the destination object.

In addition to the capabilities described above, the set of capabilities *OR* also contains the permissions from set P. Object *T1* contains the p_1 capability. Object *T2* contains the p_2 and p_3 capabilities. Object *T3* contains p_5 and p_6. *T4* contains p_7. *P3* contains p_8, and lastly, object P contains the capability p_4. This means that anyone in possession of a capability for object S can acquire and use any of the capabilities described above in a manner analogous to role inheritance.

9 Conclusion

In this paper we have presented a formal model of Role Based Access Control which is derived form the Access Control Matrix. This model is derived from the initial work of Harrison, Ruzzo and Ullman on access control models. Such a model places RBAC in relation to the traditional access control models and enables comparisons to be made between systems based on the various



(a) Role Hierarchy

	O_1	O_2	O_3	O_4
S				
S3				
T1	r			
T2	r, w, x	w, x		
T3			r, w	x
T4				w, x
P3				r
P		r		

(b) Access Matrix

Figure 3: The project role hierarchy

models. In the process we have demonstrated fundamental similarities between RBAC and capabilities. That RBAC should be related to one or the other of the basic derivatives of the ACM should come as no surprise. The ACM is the fundamental expression of discretionary access control and access control lists and capabilities represent the two intuitive methods of viewing it. Therefore it could reasonably be expected that RBAC would show some relationship to one or the other. However, the high degree of similarity found may not have been so expected.

Understanding the relationship between capabilities and RBAC and, more distantly, RBAC and the ACM, opens the possibility of applying results known for those models to RBAC (and vice-versa). It should also simplify comparisons, such as in terms of safety analysis, between systems based on the various models.

Two broad areas of future work offer themselves. First is the examination of the implications of placement of RBAC in a taxonomy of access control models. Does its similarity to capabilities indicate that implementations of RBAC based on capabilities have promise? Can known properties of capability systems be applied to RBAC systems?

The second area further extending our formalism into other access control models, such as the Chinese Wall and other lattice based models [10].

Acknowledgements

We extend our gratitude to the anonymous reviewers whose suggestions have considerably improved this paper.

References

[1] M. Anderson, R. D. Pose, and C. S. Wallace. A password-capability system. *The Com-*

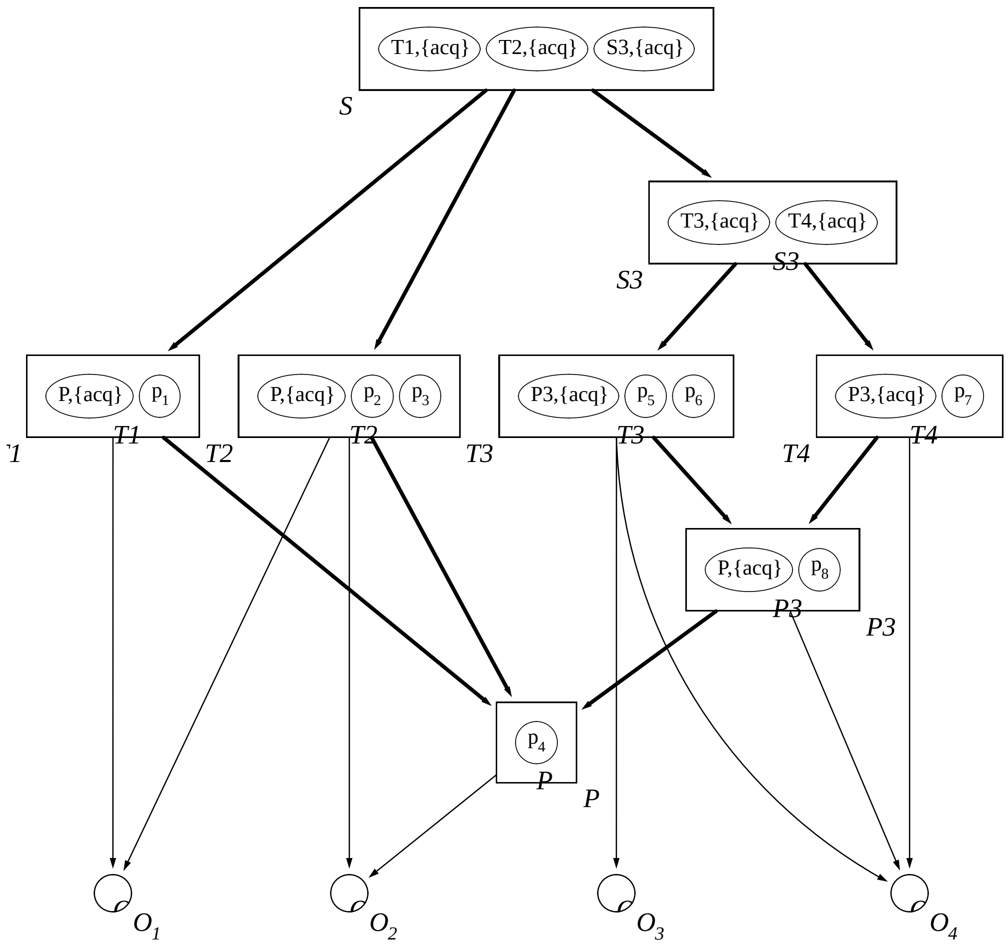


Figure 4: The example using the capability container model.

puter Journal, 29(1):1–8, February 1986.

[2] A. Dearle, R. di Bona, J. Farrow, F. Henskens, D. Hulse, A. Lindström, S. Norris, J. Rosenberg, and R. Vaughan. Protection in the grasshopper operating system. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, pages 54–72, September 1994.

[3] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[4] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–

412, July 1974.

[5] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563. October 1992.

[6] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[7] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 474–485, 1997.

[8] P. A. Karger. Improving security and performance for capability systems. Technical Report 149, University of Cambridge Computer Laboratory, Cambridge, England, October 1988. Dissertation submitted for the degree of Doctor of Philosophy.

[9] B. W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, January 1974.

[10] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, November 1993.

[11] 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.

[12] R. S. Sandhu, D. Ferraiolo, and R. Kuhn. The nist model for role-based access control: Towards a unified standard. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control*, pages 47–63, 2000.

[13] R. S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[14] G. Saunders, M. Hitchens, and V. Varadharajan. An analysis of access control models. In *Proceedings of the Fourth Australasian Conference on Information Security and Privacy*, 1999.