Delegation in the Role Graph Model

He Wang
Dept. of Computer Science
The University of Western Ontario
London, ON, Canada N6A 5B7
hewang@csd.uwo.ca

Sylvia L. Osborn
Dept. of Computer Science
The University of Western Ontario
London, ON, Canada N6A 5B7
sylvia@csd.uwo.ca

ABSTRACT

We present a model for delegation that is based on our decentralized administrative role graph model. We use a combination of user/group assignment and user-role assignment to support user to user, permission to user and role to role delegation. A powerful source-dependent revocation algorithm is described. We separate our delegation model into static and dynamic models, then discuss the static model and its operations. We provide detailed partial revocation algorithms. We also give details concerning changes to the role hierarchy, user/group structure and RBAC operations that are affected by delegation.

Categories and Subject Descriptors

D.4.6 [**Software**]: Security and Protection; H.2.7 [**Information Systems**]: Security, integrity, and protection

General Terms

Design, Management, Security

Keywords

role-based access control, delegation

1. INTRODUCTION

Delegation is an important function in many application areas, such as health care and the business world. One employee can delegate his or her job to another. The employee who receives the delegation will act on behalf of the original employee and finish the task. We call the original employee the *delegator* and the receiving employee the *delegatee*.

Delegation first received interest in the research field of distributed systems. Gasser and McDermott treat delegation as a user to system function [7]. In the role based access control (RBAC) field, there has been lots of research on delegation [5, 18, 21, 19, 3]. Most of the models use user-role assignment to perform delegation. Several revocation methods are also discussed.

We extended our Role Graph Model to support decentralized administration in 2003 [15]. In this paper we put delegation support into the Role Graph Model by using a combination of user to group assignment and user-role assignment. We divide our model into a static and dynamic model. In this paper, we focus on the static model. We also present several revocation methods, including timeout, source dependent cascading revocation, partial revocation etc. and give detailed algorithms for their implementation.

The paper is organized as follows: Section 2 reviews delegation models for access control, Section 3 presents the extended role graph model with decentralized administration, Section 4 introduces delegation into the role graph model, and Section 5 contains discussion and conclusions.

2. DELEGATION MODELS

There are several delegation models for RBAC. In 2000, Barka and Sandhu presented a framework for their first notion of delegation [4]. Their role-based delegation model, RBDM0, is based on RBAC96, and uses a user to role assignment approach supporting user to user delegation [5]. RBDM0 is total delegation, i.e. the delegator delegates all the permissions in a role to a delegatee by user to role assignment; then the original user of the role assigns a delegatee to the role. Revocation is done by timeout and by grant-independent revocation. The authors also extend the model to support partial delegation and two-step delegation by defining two different types of permissions in a role: delegatable permissions and non-delegable permissions. The delegatee can only have delegable permissions. In their second model, RBDM1 [6], Barka and Sandhu added role hierarchies and source dependent cascading revocation, which is done automatically along the delegation chain.

Zhang, Oh and Sandhu presented a new permission-based delegation model (PBDM) in 2003 [21]. This model fully supports partial and multi-step delegation. It consists of three sub models PBDM0, PBDM1 and PBDM2. They are based on the the RBAC96 model and use user to role assignment to do delegation. PBDM0 has two shortcomings: the user can delegate any permission to a delegation role. This will cause security problems if limited users collect permissions via multiple delegations. The other shortcoming is that this model only supports user to user delegation. PBDM1 solves the first shortcoming by further separating the roles into three different sets. PBDM2 is designed to support role to role and permission delegation. It divides roles to four

SACMAT'06, June 7–9, 2006, Lake Tahoe, California, USA.

different sets. The complex formal definition is given in [21]. The PDBM models can support multi-step delegation, but they neither support constraints in delegation, nor delegation in distributed environments.

Zhang, Ahn and Chu extended the RDM0 to a new model called RDM2000 in 2001 [18]. They described a prototype system for a health care environment in [19], and one for law enforcement agencies in [20]. The RDM2000 model supports hierarchical roles and multi-step delegation, which are not supported in RDM0. They also specified a rule-based language to describe the policies of RDM2000. Revocation is separated into two categories: revocation by duration-restriction constraints and user revocation. The first uses time constraints to enforce revocation. The second is done by a user. Ahn's group published three papers recently for access control in a collaborative environment [2, 1, 13] using this delegation model. The rule-based approach is very powerful for constraint enforcement. However it only considers the regular user to user delegation; it does not support administrative user delegation.

Most delegation models are centralized. Yin et al. have discussed a decentralized delegation model with management domain and trusted scope ideas for distributed systems [17]. The model divides access control in large, distributed systems into two levels: the management level and the request level. At the management level, the system consists of *Multi-centric management* which has its own authorization management domain. At the request level, regular users make a *Cascaded Request* which requires more than one service to respond to the request. They classified delegation into two levels corresponding to the levels mentioned above. At the authorization management level, the delegation is called delegation of authority. At the request level, the delegation is called delegation of capability.

Wainer and Kumar considered different constraints that can be applied to RBAC delegation and presented their fine-grained user to user delegation model in 2005 [14]. This model distinguishes two types of access rights: object rights and delegation rights with constraints. It uses user to role assignments to do delegation. The revocation is source dependent cascading revocation similar to System R with an improved algorithm. This model also has another revocation method, "revocation with downgrade", which tests and updates the depth for cascading revocation. An extension of this model is time-restricted delegation which uses timeout to revoke the delegation.

Atluri and Warner studied delegation in workflow management and introduced a conditional delegation model [3]. This model introduces several constraints (conditions). The conditions include time intervals, workload limitations and task attributes. The constraints are also divided into four different types: authorization constraints, delegation constraints, task dependency requirements and role activation constraints. The constraints are defined as rules of a logic program. There are three kinds of conditions for delegation. A *temporal delegation condition* is a condition on the delegation start time and/or the time interval of the delegation. A *workload delegation condition* is a condition of a specific workload level. *Value delegation conditions* control a delegation by attributes. Several rules are defined to support conditional delegation. Some constraints can be verified before the execution of workflow and some must be verified and enforced during workflow execution. The authors call this verification *delegation consistency*; the former is called static consistency and the later is called dynamic consistency. Checking steps are also discussed.

Most delegation models are based on user to role assignment. In these models, to support partial and role-role delegation, the role hierarchy has to be modified to a very complex structure. Cascading revocation is also very complex. This approach works better only for total delegation, and limits the implementation and usefulness of partial and role-role delegation. We use a combination of user to group assignment to do partial and role-role delegation, and user-role assignment to do total delegation. Our model has minimum impact on the role hierarchy and overcomes the shortcomings of the user to role assignment approach. Our goal is to provide a simple and easy-to-use delegation model and the administration tools to support it.

3. THE ADMINISTRATIVE ROLE GRAPH MODEL

The role graph model [10, 11] has three components: users, roles, and privileges. The original model assumes that the administration of a role graph is centrally controlled by a system administrator. In [15], we extended this model to support decentralized administration by introducing the *administrative domain* concept.

3.1 Basic Role Graph Model

As shown in Figure 1, the left plane contains the group graph, which shows user-group memberships. A group is a set of users who can be considered as a unit, e.g. a department or a committee. Group membership can also be based on user credentials or attributes. This graph has a hierarchical structure. The relationship between a lower group and a higher group is set containment. If there is an edge or path from group g_i to g_j, then g_j contains all the members of g_i.

The role graph or role hierarchy is in the middle. The nodes represent roles, r, which consist of a name, r_{name} and set of privileges, r_{pset}. The edges $r_1 \rightarrow r_2$ show the *is junior relationship*; when $r_1.rpset \subset r_2.rpset$, we say r_1 is junior to r_2, denoted by $r_1 < r_2$. We also say r_2 is senior to r_1. The privileges of a role are divided into two sets. The *direct privileges* are directly assigned by the administrator. The *effective privileges* of a role are the union of its direct privileges and the privileges inherited from its junior roles.

The role graph has the following properties:
1. It has a MaxRole, which contains in its r_{pset} the union of all the privileges of the graph. MaxRole does not need a user assigned to it. (It is used as a summation of the privileges).
2. It has a MinRole, which contains the minimum set of privileges available to all roles in the system. It can be the empty set.
3. The role graph is acyclic.
4. Every role has a path from MinRole.
5. Every role has a path to MaxRole.
6. For any two roles, r_i and r_j, in the graph, if $r_i.rpset \subset r_j.rpset$ then there must be a path from r_i to r_j.

On the right side is the privileges plane. Each privilege is a pair (o, m), where o is an object and m is an access mode of the object o [9]. The privileges have privilege-privilege implication relationships. When a privilege is assigned to a role, we require that the implied privileges must also be granted. For example, if a user has read privilege on a table

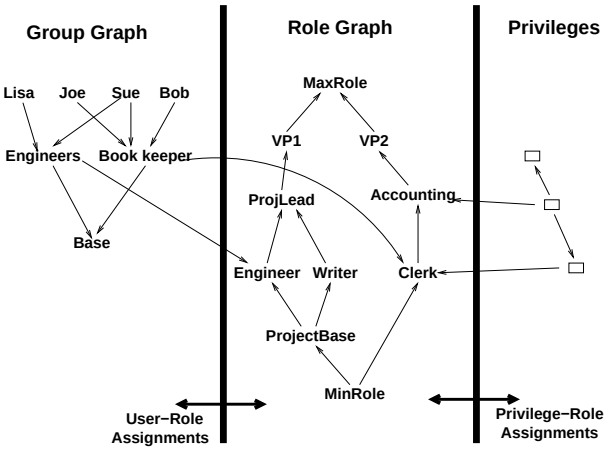

Figure 1: The Role Graph Model

in a database, the user also has the privilege to read all the attributes of this table. Details are given in [9].

There are five authorization relationships in the role graph which ultimately affect a user's ability to have a privilege:
1. privilege-privilege relationships.
2. the assignment of privileges to roles.
3. role-role relationships.
4. the user-role assignment.
5. the edges in the group graph.

It is up to the security administrators/designers to create appropriate authorizations so that users can carry out their required tasks.

The user-role assignment is the most important relationship for this paper. When we assign a group to a role, all privileges of the role are assigned to the group. Since the group contains subgroups and users, all the subgroups and users also get the privileges of the role implicitly through group membership. If we add a new subgroup to the group, then the subgroup will get the privileges of the role(s) that the super group is assigned to. From Figure 1, we can see Lisa and Sue are the members of the engineers group. If the engineers group is assigned to role engineer, then all members of the engineers group, including Lisa and Sue, will have the privileges of the engineer role. We will use this property to do our delegation.

In order to administer access using the role graph model, several algorithms have been developed, implemented, and tested. Most of them deal with role graph management: role addition and deletion, edge addition and deletion, permission addition and deletion [11]. Some involve assignments: user to role assignment, privilege to role assignment, or privilege to privilege implications [12, 9].

3.2 Administrative Domains and Administrative Roles

An administrative domain is an administrative unit assigned to an administrator. It is a role graph which has all the basic components of a regular role graph model. There are two types of administrative domains: the *default domain* and the regular domains. The default domain represents the whole organization, and is administered by the system security officer (SSO). A *regular domain* contains all the junior roles of a unique role except MinRole. This unique role is on the top level of the role hierarchy of the domain. We use its name to identify the domain, so we call it the *domain identifier* or *domain ID*.

An administrative domain is a subset of the role graph. Let us denote by D an administrative domain which contains a set of roles. r_D is the domain identifier of D. Let R be the set of roles of the organization. Then we have $r_D \in D$ *and* $D \subseteq R$. The roles in the domain are given by $\{s | s < r_D$ and $s \neq MinRole\} \cup r_D$, where "$<$" is the *is junior* relationship.

The users and groups assigned to the roles of an administrative domain form the user/group plane of the domain. Similarly the privileges assigned to the roles of this domain also form the privilege plane of the domain. We can see the administrative domain is a basic role graph except for MinRole. The domain identifier is the MaxRole of the domain's role hierarchy.

An administrative role is a triple $(name, pset, D_{rset})$. It has role name, set of administrative privileges($pset$) and a set of roles which form the administrative domain (D_{rset}). The administrative domain is administrated by the user(s) assigned to this administrative role.

The relationship between administrative domains and administrative roles is a many to many relationship. The administrative domain can be managed by one or more administrative roles. The administrative role also can manage one or more domains.

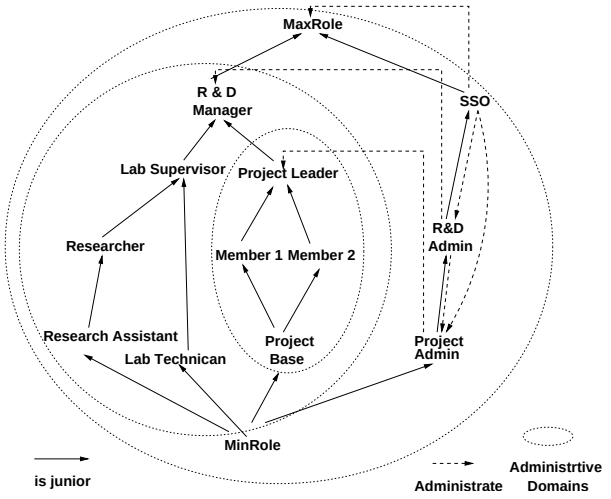

Figure 2: A Role Graph Showing Administrative Domains for the R&D department

Figure 2 shows an example role graph with administrative domains. The domain whose domain ID is R&D manager is assigned to R&D admin. All its junior roles except MinRole are the elements of the D_{rset} of the R&D administrator. The R&D administrative domain contains the project domain. The D_{rset} of the project domain $\subset$ D_{rset} of R&D domain. The domain identifier of the project domain is Project Leader.

Roles of administrative domains can not overlap arbitrarily. The Users and Groups of administrative domains can overlap since users or groups can work in different domains.

Privileges of administrative domains can also overlap. An administrator assigned to an administrative role can only administer the roles in their administrative domain. If there is an operation that needs to reach outside of the domain, the operation has to be performed by the administrator of a surrounding domain. The administrative roles are part of the role hierarchy. A detailed discussion of administrative operations, algorithms of this model and the comparisons with Crampton's model can be seen in [15].

4. ROLE GRAPH MODEL FOR DELEGATION

In our discussion of delegation for the administrative role graph model, we will distinguish between design time and run time. The design of the role graph, group graph and user role assignment, etc. would be performed before the system is deployed. It might also be that from time to time, the system is brought down for upgrades. Any time in which the system under control is not running, we will refer to as *design time*. The system administrators should anticipate, at design time, which parts of the system are important enough to warrant delegation if certain users are not at work. It may be that certain privileges or roles should never be delegated, but that others can be. Any real unanticipated requirements for delegation not part of the static design, can be added at run time by dynamic delegation. This is the basis for our separation of the delegation task into static and dynamic models.

4.1 Static vs. Dynamic Model of Delegation

Although delegation happens at run time, many of the requirements can be predicted at design time. A well-defined delegation design can achieve better results at run time. Thus we separate our model into static and dynamic models.

The static model consists of the role graph model that is being developed in the design phase of the system for offline testing or during the modification of a running system. In this phase, the administrator can use the model to design and test the users, roles, privileges and their relationships in the system. The static model has the following advantages:

- In the static model, the configuration of access control can be tested and tuned without affecting the real running production system.
- In the static model, conflict of interest and other constraints can be defined and tested to suit the needs of the run time system.
- The leaking of security due to bad design can be prevented by specially designed algorithms and simulation of the run time system; thus the designer can catch these errors without shutting down the system.

The dynamic model controls the role graph model at run time. Some errors can only be found at run time. The user activates their assigned and delegated roles to perform their tasks in a running session. Delegation in the dynamic model can be ad-hoc without the administrative intervention. The user to role assignment and privilege assignment also can be changed dynamically according to the job task. Some reference monitor controls and enforces access control policies that are defined in the static model. Error detection, correction and avoidance are also important in this model. For better control and performance analysis, an auditing subsystem is also needed in the dynamic model. The dynamic model is a powerful and flexible model.

The static and dynamic models cannot be used on their own. The static model is the basis for the dynamic model. It defines the access control policies of the system, and predicts and corrects the errors that may happen in the dynamic model. The dynamic model is the enforcement of the static model. It also provides feedback to the administrator concerning future modifications of the static model.

4.2 A Static Model of Delegation

We will start with an an example, and then give formal definitions. First, we need to define several new components:

- *Delegator role*: a delegator role is a special administrative role junior to the domain administrator role. The privileges of this role are create-subrole, assign privileges and user-role assignment and revocation. The create-subrole privilege allows the user to create a special sub-role that is junior to a regular role to which the user is assigned. The special sub-role is called a delegation role to be defined shortly. The user-role assignment/revocation privilege allows the user to assign other users/groups to the delegation role and revoke them later on.
- *Delegation role*: a delegation role is a special role in the role graph. The delegation role privilege set $rpset_d$ has relation with the privilege sets of the delegator's regular roles $rpset_r$: $rpset_d \subset \bigcup rpset_r$. It not only can be created by the user at run time, but also can be created by the domain administrator at design time. Depending on the privileges set, it can be anywhere in the role graph. (The role graph algorithms will place any role between its juniors and seniors according to its *pset*).
- *Delegation edge*: a delegation edge is a labeled edge from a delegatee in the user/group plane to the delegation role and between delegatees. This edge is created by the delegator when assigning a delegatee to a delegation role or to another delegatee group. The label on the edge has three components (t, d, C): $t = timestamp$ is the expiration time of this delegation; $d = depth$ is the depth of further delegation allowed. If it is $*$, then we allow unlimited delegation. C is a set of constraints on the delegation.

Figure 3 shows an example of the extended model. User Alice is a project leader and also assigned to the delegator role. She wants to delegate all of her job to user Bob who is a member of the project. Alice can create a delegation edge and assign Bob to be a member of her own group in the user plane, and mark the labels on the delegation edge. Since the junior user is a member of a senior group and inherits the privileges of the senior user, Bob now has the privileges that Alice has and can perform the task. We use the nature of inheritance in the user/group plane to achieve the user-to-user total delegation. The delegation also can be created by the project administrator. In this case, the delegation result can be tested at design time, thus we can prevent conflicts or security problems. This is a major advantage of the static delegation model. When Alice wants to do delegation, she

simply just assigns Bob's group to be a subgroup of her (Alice's) group.

The above example will not work for partial delegation, because the user will inherit all the privileges in the roles assigned to the senior users or groups. We use user-role and user-user assignments to do partial delegation. Figure 4 shows an example of partial delegation. Now Alice wants to delegate part of her job to user Bob who is a member of the project. Alice can create a delegation role, which has some privileges needed for the job task. The privilege set of the delegation role is a subset of the privileges of the project leader role, so the delegation role is a junior role of the project leader role. After creating the delegation role, Alice can create a group DELEGATEE, assign DELEGATEE to the delegation role, then make Bob's group a subgroup of DELEGATEE and mark the labels on the delegation edge. Bob now has the privileges in the delegation role and can perform the task. In this case, Bob is not assigned to Alice's group directly; otherwise he will get all the privileges from Alice by inheritance. That is why we need to create group DELEGATEE and assign Bob to it. Bob also can further delegate the task to Carol by making Carol's group a subgroup of his own group, giving multi-step delegation.

The above is an example of user-to-user delegation. We can also achieve permission to user delegation without changing the structure. The domain administrator can put one or more permissions into the delegation role and assign the DELEGATEE group to it, giving permission to user delegation.

We also can use the same structure to achieve role-role delegation. The domain administrator or delegator can create a delegatee group and assign it to the regular roles of the delegator; then make the users or groups that should be assigned to the delegated role members of the delegatee group. We get the same result as role to role delegation by using user/group assignment. Suppose we want to delegate role r_2 to role r_1. Then r_2 is the delegated role. The users U_{r2} that are assigned to r_2 are the delegatee of the delegation. After we make all the users in U_{r2} members of the delegatee group which is assigned to role r_1, the users of U_{r2} can perform the delegated task of r_1. Thus the role-to-role delegation is achieved.

This simple model is very powerful. It can support user-to-user, permission-to-user, role-to-role delegations in RBAC.

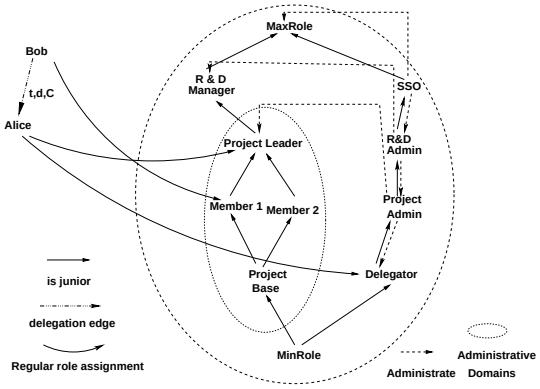

Figure 3: An Example of Total Delegation in the Extended Role Graph Model

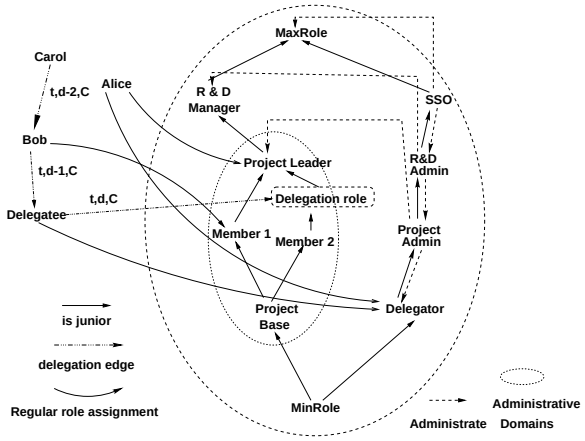

Figure 4: An Example of Partial Delegation in the Extended Role Graph Model

As shown in the above example, we need to add some components into the role graph model:

- The Delegator, $u_{dr} \in U_{dr}$, is the issuer of the delegation. The Delegatee, $u_{dt} \in U_{dt}$, is the receiver of the delegation. The delegator cannot be their own delegatee at the same time: $u_{dr} \neq u_{dt}$. This prevents the delegator from refreshing the delegation by assigning herself to the delegatee. The delegatee receives the access privileges from the delegator. The delegatee is a special group which contains at least one user. It also can have other delegatees as sub groups. We use delegatee u_{dt} for a single user and U_{dt} for a delegatee group or set of delegatees. The set of delegators U_{dr} and the set of delegatees U_{dt} form a delegation users/groups set U_d, thus $U_{dr} \bigcup U_{dt} = U_d$ and $U_d \subseteq U$.
- Delegation role: $r_{dt} \in R$ and $r_{dt} \subset \bigcup R_{rgdr}$, where r_{dt} is a delegation role and R_{rgdr} is a set of regular roles to which the delegator is assigned. R is the regular role set and $R_{rgdr} \subset R$. As far as privilege sets are concerned, $r_{dt}.rpset \subseteq \bigcup r_{rgdr}.rpset$; thus the delegation role can be anywhere in the role graph. Since the delegation role is a regular role and $R \cap R_{admin} = \phi$, the delegation role is not a delegator role.
- Delegator role: $(r_{dr}, \{c_{sub}, ura\})$ is an administrative role, where c_{sub} is the create-subrole privilege which allows the user to create a delegation role. $ura = \{(u_d, op) | (u_d \in U_d \ and \ op \in \{assign, revoke\})\}$ are the user-role assignment and revocation privileges which allow the delegator to assign or revoke the delegation users/groups U_d to/from the delegation role.
- Delegation edge: $e_d(t, d, C)$ is a labeled edge from the delegatee group to the delegation role and between delegatees. The labels on the edge are: $t = timestamp$ is the expiration time of this delegation; $d = depth > 0$ *or* $*$ is the depth of further delegation allowed. If it is $*$, then we allow unlimited delegation. C is the constraint set for fine-grained control of delegation. C is of the form $c_1 \wedge c_2 \wedge ... \wedge c_n$ where the c_i are Boolean expressions; if C is true then the delegation is not allowed.

- Delegation path: $U_{dt} = u_{dt1}, u_{dt2}, ..., u_{dti}, u_{dtj}, ..., u_{dtn}$ is a path in the user/group plane that is formed by the delegatee groups who have sub-group relationships. The depth d in the label of the delegation edges is decremented by one between two directly connected delegatees: $d_j - d_i = 1$. This path represents a delegation chain.

We will discuss the operation and administration of this model in section 4.2.3.

4.2.1 Constraints and Rules

Delegation is very powerful and must be finely controlled. There are several kinds of constraints possible.

Delegatee constraint is a constraint to specify the groups as the delegatee. Employees have different backgrounds and skills, so only small groups of employees may be qualified to perform the delegated task. In the static model, we use U_{dt} to specify the delegatees. For every delegation, we can predefine a set of users in the group U_{dt}, such that the delegator can only assign the user $u \in U_{dt}$ to the delegation role. In this way, we can control that the delegatee is qualified for the delegated task.

Delegated role constraint controls the role-to-role delegation. It is similar to the idea of the delegatee constraint. Only the users or groups that are assigned to the delegated role can be the delegatee. Suppose we want to delegate role r_2 to role r_1, then r_2 is the delegated role. As mentioned in the previous section, we can use user-role assignment to do the role-role delegation. The users/groups u_{r2} that are assigned to r_2 are the qualified delegatees. So we can put u_{r2} into U_{dt} for delegation.

Delegation role constraint controls which roles are allowed to be delegated. In a business, some roles are critical due to the nature of the responsibility of the role or due to the skills of the employee who is assigned to that role. There may be no other qualified employees to do the task. So these roles cannot be delegated. We can enforce this in the other direction by defining allowed delegation. If a regular role r has the user u who is assigned to a delegator role $r_{dr} \in R_{admin}$, then this role can be delegated, because the user has the privilege to create a delegation role. By default, all the roles without this property are not delegatable.

Maximum privilege set constraint, c_{pmax}, controls the maximum set of privileges allowed by delegation. The delegatee has the privileges set $P_r \bigcup P_{dt}$. This constraint controls the privileges after delegation, and requires that $P_r \bigcup P_{dt} \subseteq c_{pmax}$. For example, if several users delegate their privileges to one user, then this user would have all the privileges of those users and would become very powerful. This may cause a security problems, so we need to control it.

Absence constraint, c_{ab}, controls that the delegation can only happen when the delegator is not at work. This constraint is a simple Boolean variable and $c_{ab} \in C$.

Workload constraint, c_w, controls that the delegation only happens when the workload exceeds a certain level. When the workload of the delegator w is higher than a predefined level w_l, then the delegation can be performed. $c_w \in C$.

Location constraint, c_l, controls that the delegation only happens at predefined location LC, such as workstations of the delegator or the computers in the same room as the delegator. $c_l \in LC$ and $c_l \in C$.

A separation of duty constraint, $c_{sod} \in C$, enforces static separation of duty in the role graph. If two roles are defined to conflict, then the users who are assigned to these roles can not be delegated to each other.

We can group different constraints into several categories. The delegatee constraint and delegated role constraint are membership constraints. They control the membership of the delegatee. The delegation role constraint and maximum privileges constraint are RBAC delegation constraints. Absence, location, workload constraints and separation of duty constraints are environmental constraints. They control the delegation based on the related job task.

4.2.2 Constraint Enforcement and Error Prediction

Constraints can be defined and tested at design time. When a delegation is added to the system, a series of tests are performed to test whether the delegation meets those constraints. If a delegation causes the system to be insecure or conflict with the policy, then we say an error has occurred. We can use several testing functions to test the delegation at design time and predict the result of the delegation. Thus we eliminate problems as much as possible at design time.

To test membership constraints, we define a Boolean function $isDelegatee(u_{dt})$. If the delegatee $u_{dt} \in U_{dt}$ of a delegation role, then this function returns $true$; otherwise it returns $false$.

For a delegation role constraint, we define a Boolean function $isDelegatableRole(u_{dr}, r)$. If the delegator u_{dr} is assigned to r and $r_{dr} \in R_{admin}$, then this function returns $true$; otherwise it returns $false$.

To enforce the maximum privileges constraint, we define a Boolean function $notExceedMax(u_{dt})$. If $u_{dt}.P_r \bigcup P_{dt} \subseteq c_{pmax}$, this function returns $true$; otherwise it returns $false$.

The absence constraint is not tested at design time; it is tested at run time. The administrator or manager can set this Boolean variable to true at run time to allow the delegation. While not tested at design time, this constraint is defined at design time. If an administrator wants to use this constraint, she can simply add it to the constraints set C of the delegation edge. Similarly, the workload constraint is not tested at design time. The administrator only puts the predefined workload level w_l in the constraint. The constraint is checked at run time.

Finally, Figure 5 gives an algorithm which checks the constraints and returns true if they are all met. When we create a delegation, this algorithm is called. If it returns true, the delegation is allowed and is created in the system; otherwise the delegation is not created.

4.2.3 Administration of the Static Model

Delegation can be created by a user or by an administrator. The former is self-directed delegation, but it still requires the administrator to create the delegator role and assign the delegator to it. This can have better control than the traditional discretionary access control. The delegation also can be administered by the administrator. We will use partial delegation to describe the operations in this section.

When creating the delegation, the delegator or administrator creates the delegation role and delegatee group, tests all the constraints, then assigns the delegatee to the delegation role. The detailed algorithm is given in Figure 6.

From the algorithm createDelegation, we can see that this operation supports both total and partial delegation, de-

Algorithm canDelegate(RG, r, u_{dr}, u_{dt})
input: a role graph RG, the regular role r, a delegator u_{dr} and delegatee u_{dt}
output: true if the delegation is allowed, otherwise false.
begin:

```
result ← false
get c_sod of u_dt
if r is in c_sod or w < w_l or c_l ∉ LC or c_ab is false
        result = false
else
        result ∧ = isDelegatee(u_dt)
        result ∧ = isDelegatableRole(u_dr, r)
        result ∧ = notExceedMax(u_dt)
return result
```

end

Figure 5: Algorithm canDelegate

Algorithm createDelegation($RG, r, r_{dr}, u_{dr}, u_{dt}, C, t, d$**)**
input: a role graph RG, the regular role r, the delegator role r_{dr}, a delegator u_{dr}, delegatee u_{dt}, constraints set C, expiration time t and delegation depth d
output: the role graph with delegation created.
begin:

```
if u_dr is not assigned to r or not canDelegate(RG, r, u_dr, u_dt)
   abort
otherwise
   P_list ← [ ]
   P_list = get all effective privileges from r
   let user u_dr choose P_dlist ← P_list
   if P_dlist ⊂ P_list
      addRole(RG, r_dt, P_dlist)
      create and assign delegatee u_dt to delegation role r_dt
      label the edge from u_dt to delegation role r_dt with t, d, C
      if d > 1, then
         assign delegatee u_dt to delegator role r_dr
   else
      assign delegatee u_dt to delegator u_dr
      label the edge from u_dt to u_dr with t, d, C
return RG
```

end

Figure 6: Algorithm createDelegation

pending on whether the delegator or administrator chooses all the privileges of the regular role or a subset.

Algorithm createDelegation implements single step delegation. We can modify it to support multi step delegation in the following way. The delegation depth is controlled by the label d, so we need to test that the label d from the delegator is greater than 1; if $d > 1$, then further delegation is allowed. We can create delegation by calling the createDelegation algorithm and pass in the parameter $d = d - 1$. The results of this operation are: a delegation edge is created and labeled with $t, d-1, C$. In this way, multi step delegation is achieved very efficiently. All the delegations from a multi step delegation form a chained delegation, which we call a delegation path. The depth component in the label of the edges on the path is controlled by d in every step.

Delegation with transfer of authority is another feature of discretionary access control (DAC) with change of ownership. In DAC, an owner can transfer authority to another user when she delegates access. After the delegation, the delegator loses the access rights to the object; the delegatee gets full power over the object. Our model can support this operation very easily. Because the delegator has assign/revoke privileges from the delegator role, we can change the algorithm slightly. After the delegator creates the delegation, she can revoke herself from the regular role and from the delegation role. After this operation, the user role assignment edges from the delegator to regular role r and delegation role r_{dt} are deleted. The delegator will not have access to the role; the delegatee has full control over the role r and r_{dt}.

Revocation of delegation is very important. In order to provide more choices to the user, we support timeout revocation, constraint violation revocation, cascading revocation and partial revocation in our model. The basic operation of revocation is deleting the delegation edge, after which the delegation no longer exists.

Time-out revocation is enforced by the system. The system will check the expiration time in the label of the delegation edge. If it is expired, the delegation has to be revoked and the delegation edge is deleted from the system. Depending on the system setting, if the system is configured to use cascading revocation, then all delegations on the delegation path are revoked at the same time. If the system is not using cascading revocation, then only the expired delegations are revoked.

Similar to timeout revocation, constraint violation revocation is also enforced by the system. If a constraint in C is violated, then the delegation is revoked. This revocation also can lead to a cascading revocation depending on the system setting.

Cascading revocation is more complex. Because a delegation may originate from different delegators, we only want to revoke the delegation that originates from the revoker. This is called *source dependent revocation*. In order to do it correctly and efficiently, we will use the labeling on the delegation edges to do cascading revocation. We start from the revoker and get the label of the delegation edge. From the label, we get the depth. We delete this depth d. If there is no other depth left, we delete the edge. The first revocation is done. Then we find the delegation edge that starts from the delegatee and points to the next delegatee. We get the label again, and delete any edge whose depth is equal to the first delegation depth minus one. If the delegation originates from one source, then there is no alternate label with $d \leq 1$ left. We delete this edge. Otherwise we continue on until the delegation path is processed. The detailed algorithm can be seen in Figure 7.

Figure 8 shows an example of cascading revocation with limited depth. With the exception of the Delegation Role, all the nodes in this graph represent users or groups in the group plane. An edge from user node D to user node B represents a delegation from user B to user D, as this is accomplished by creating an edge in the group graph making D's group a subgroup of B's group. We can see that the revocation of $B \rightarrow D$ causes a cascading revocation $D \rightarrow E \rightarrow F$. The depth of each step is decremented by one. So we can use this property in our algorithm. When we perform cascading delegation, we trace the labels of the delegation edges to find the decrementing depth that originates from the delegator; then we can delete this label, the delegation is revoked. When there is more than one delegator, our algorithm can only revoke the delegation from the revoker. In the example, $D \rightarrow E$ has two sources which have two

Algorithm cascadingRevocation($RG, r_{dt}, u_{dr}, u_{dt}$)
input: a role graph RG, the delegation role r_{dt},
a delegator u_{dr}, and delegatee u_{dt}
output: the role graph with delegation revoked.
begin:
get the label l of edge $u_{dr} \rightarrow u_{dt}$
$n = l.d$
$U_{list} \leftarrow \{u_{dr}\}$
$dr = 1$ //number of delegators to be processed
while $n > 0$
For the beginning dr delegators in U_{list}
$E_{list} \leftarrow$ edges that start from the delegator u
while E_{list} is not empty
For every label l of e *in* E_{list}
//delete all following edges that have n=depth
if $l.d == n$
delete l
dr ++
$U_{list} = U_{list} \bigcup \{$delegatee of $u\}$
if $l == null$
delete edge e
else
remove edge e from E_{list}
$dr - -$ //since dr initially 1, adjust it
remove u from U_{list}
if $n \neq *$
$n = n - 1$
//if revoker is delegatee group, delete delegation role
if u_{dr} is the delegatee and no out edges
delete delegation edge from u_{dr} to delegation role r_{dt}
delete delegation role r_{dt}
return RG
end

Figure 7: Algorithm cascadingRevocation

depth labels. We only delete one label with $d = 2$, since the previous label on our revocation chain had $d = 3$. Another label $d = 1$ is untouched, because it is from another source C. Thus, source dependent revocation is achieved. If there are two delegators that have the same depth, then there will be two identical numbers in the labels of the delegation edge. We can simply just delete one of the identical labels.

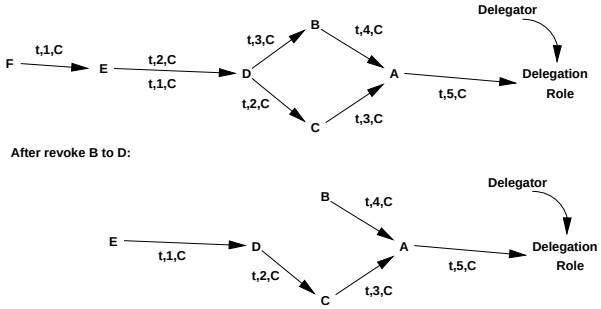

Figure 8: An example of cascading revocation with limited depth

When there is unlimited depth delegation, the label of the depth is $*$. There is no decrement by one property from delegator to delegatee. Instead, every step is unlimited delegation. In this case, we can just delete the edges that have their origin at the delegator and depth is $*$. If there are two delegators delegating to the same delegatee, then there will be two labels that have $*$ as the depth but the starting points are different. We can delete the one from the revoker, then the result is source dependent cascading revocation. Figure 9 shows an example of this idea.

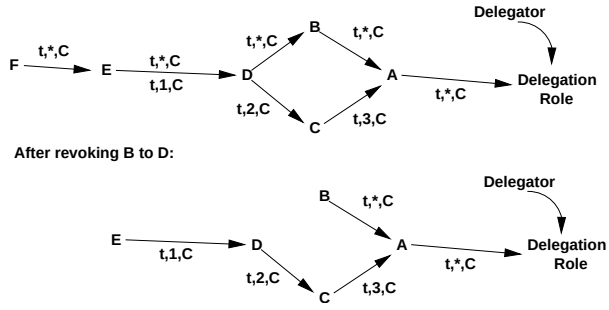

Figure 9: An example of cascading revocation with unlimited depth

The delegation depth also can be changed from unlimited to limited. In this case, the depth will change from $*$ to an integer. The other labels will either have depth with decrement by one property if it is limited delegation, or have $*$ depth as unlimited depth delegation. If there is one depth in the labels that has integer depth but does not have decrement by one property from the previous delegator, we can delete this label to achieve source dependent cascading revocation. This is shown in Figure 10. From B to D is unlimited delegation; D changes the depth of the delegation to two and delegates to E; E also has limited delegation from $C \rightarrow D$. We can see $C \rightarrow D \rightarrow E$ has the decrement by one property, but $B \rightarrow D \rightarrow E$ does not. So we can delete the label with depth 2 from D to E, giving source dependent cascading revocation.

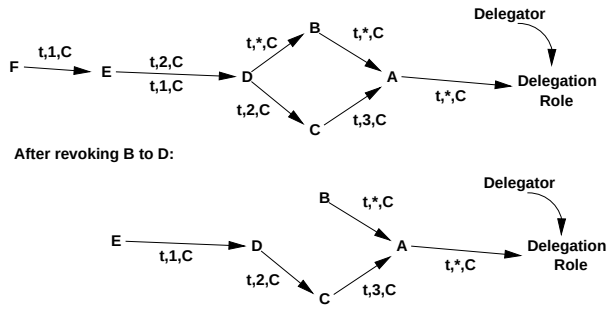

Figure 10: An example of cascading revocation with changing of depth

Most delegation models do not support partial revocation. In the business world, partial revocation is necessary. There are several different kinds of partial revocations: partial privilege revocation, partial delegatee revocation and partial delegation path revocation. In our model, we put partial revocation into consideration and present several partial revocation algorithms.

Partial privileges revocation requires some privileges to be revoked from the delegation role. For example, a delegator delegates a task to a delegatee. When the delegator finds the job is behind schedule, the delegator may want to split

the delegated task to other delegatees. This requires that the delegator partially revoke some privileges from the old delegatee and delegates them to a new delegatee. We can delete a subset of privileges from the delegation role. The remaining privileges are still available to the old delegatee. Thus $P_{revoke} \subset P_{dt}$.

Partial delegatee revocation is the case that some of the users in a delegatee group need to be revoked. This is a user revocation from a delegatee group, not a delegatee revocation from a delegation path which will be discussed in the following paragraph. For example, the manager has delegated the project developer group to a testing job. After the job is half finished, the manager wants some developers to do a system developing job. The rest of the developer group will continue with the testing job. We can simply achieve this revocation by revoking the users from the group. This operation is available as the original RBAC user -group assignment/revocation operation.

The most useful and interesting partial revocation is the partial path revocation, where we only want to revoke several delegatees along the delegation path instead of complete cascading revocation. We can divide this kind of revocation into several kinds based on the location of the revocation. We can partially revoke the beginning, the end or the middle of the path. We also can revoke several non-connected delegatees in the path. This makes the revocation very complex. We need to design a good algorithm for this revocation.

The algorithm is shown in Figure 11. We can see that the depth is decremented by one along the delegation path. A delegation edge is a labeled edge, so we can simply update the label that follows the revoked delegatee by decrementing the depth by one along the rest of the delegation path. In this way, partial revocation can be performed no matter the position of the revocation. The algorithm works in all situations, such as at the beginning, the end, and in the middle of the path or for several non-connected delegatees in the path. Partial revocation also provides a useful operation: *delegation path shrinkage*. A delegation path can be very long if all delegators further delegate to other users. In most cases, the delegators in the middle of the path are not willing to perform the delegated task; otherwise they will not delegate to others. This gives a long delegation path that is difficult to manage. The effective delegation in the path is the delegator at the beginning and the delegatee at the end of the path. So if we can partially revoke the delegatees in the middle, the result is a shorter delegation path which contains the users who are willing to perform the delegated task. The shortest delegation path is called an effective delegation path. This operation is called delegation path shrinkage.

All of above are the administration of delegation. We also need to consider the regular administration of RBAC that is affected by delegation. Some administrative operations will not be affected by delegation. Adding privileges and assigning users are this kind of operation. When we add a privilege to a regular role, the delegation role is not affected, so this operation is safe under delegation. A similar idea applies to user assignment and editing a privilege. Since the privilege is unique in the mode, changing a privilege will affect both the regular role and the delegation role, so we do not need to perform an extra operation when editing a privilege. All other RBAC operations are affected by delegation. These operations are as follows:

Algorithm partialRevocation$(RG, r_{dt}, u_{dr}, u_{dt})$
input: a role graph RG, the delegation role r_{dt},
a delegator u_{dr}, and a delegatee u_{dt} to be revoked
output: the role graph with delegation revoked.
begin:
  find an edge e of $u_{dr} \rightarrow u_{dt}$
  depth $n = l.d$ of e
  delete label l of e
  find the subgroup u_s of u_{dt} with $l.d == n-1$
  delete label l of e
  create delegation edge between u_{dr}, u_s with $l.d = n$
  if labels of $u_{dr} \rightarrow u_{dt}$ and $u_{dt} \rightarrow u_s$ are empty
    delete $u_{dr} \rightarrow u_{dt}$ and $u_{dt} \rightarrow u_s$
  use depth first search to update labels of following
  delegates of u_s with depth decremented by one from n
  return RG
end

Figure 11: Algorithm partialRevocation

- Delete privilege. If a privilege that gets deleted is also in a delegation role, then this privilege also needs to be deleted from the delegation role. The operation becomes partial revocation.
- Add role. If we add a role which has the same privilege set as a delegation role, then duplicate roles occur. We can let the administrator decide either to change the privileges of the delegation role, or change the delegation role to a regular role.
- Delete role. When we delete a role that has a delegation role, the delegation role also needs to be deleted. This starts a cascading revocation of delegation. The delegation role and all the delegates are deleted after this operation.
- Add edge. When adding an edge in the role graph, the set of effective privileges is updated, which may cause duplication with a delegation role. The solution is similar to adding a role. The administrator can decide either not to add the edge; or add the edge, delete the duplicated delegation role and assign the delegatee to the role which is the end point of the edge.
- Delete edge. When deleting an edge, some privileges in the effective privilege set are deleted. If these privileges are also in the delegation role, then they have to be revoked from the delegation role. This starts a partial revocation.
- Revoke a user. When we revoke a user from a regular role, we need to test the user assignment. If the user is a delegator, we need to revoke the delegation path that starts from the user. Thus, the cascading revocation algorithm is called. If the user is a delegatee, then all the delegatees who follow her in the delegation path need to be revoked. So the end of path partial revocation is performed in this situation.

From the above discussion, we can see delegation affects most of the regular RBAC operations. The administration of RBAC also needs to be modified to reflect delegation. All the changes are simple; we can modify the algorithms easily according to the above discussion.

5. DISCUSSION AND CONCLUSIONS

We have presented delegation in the administrative role graph model by using a combination of user/group assignment and user-role assignment. It is interesting to note how useful the concept of the user/group graph has been in formulating a model for delegation. Together with the idea of administrative roles, this approach has several advantages.

The separation of user-group assignment and user-role assignment solves the shortcomings of the user-role assignment approach. The user/group assignment provides user to user delegation; the user-role assignment gives permission to role and partial delegation. They also provide clean visual aids for the administrator to configure delegation.

Putting the delegation path in the user/group plane gives easy understanding and powerful source dependent revocation. The revocation algorithm can be designed efficiently. The labeling of the delegation edge can be easily changed to use a timestamp, in which case the existing algorithms from System R are relevant [8, 16].

We provided a detailed partial revocation operations and algorithms which support partial privileges revocation, partial path revocation and delegation chain shrinkage. Partial revocations algorithms have not been previously discussed.

We discussed the changes to the role hierarchy structure and user/group plane that are affected by delegation. We analyzed how delegation changes the normal RBAC operations and gave solutions.

We also separated our delegation model into static and dynamic models. This makes administration tasks and implementation easier. We will discuss the dynamic model and cross domain delegation in the future.

6. ACKNOWLEDGEMENTS

This research is supported of the Natural Sciences and Engineering Research Council of Canada. He Wang's research has been supported by an OGSST scholarship.

7. REFERENCES

[1] G.-J. Ahn and B. Mohan. Secure information sharing using role-based delegation. In *Proc. ITCC 2004. International Conference on Information Technology: Coding and Computing*, pages 810 – 15, 2004.

[2] G.-J. Ahn, L. Zhang, D. Shin, and B. Chu. Authorization management for role-based collaboration. In *IEEE International Conference on Systems, Man and Cybernetics*, pages 4128 – 34, 2003.

[3] V. Atluri and J. Warner. Supporting conditional delegation in secure workflow management systems. In *Proceedings Tenth ACM SACMAT*, pages 49–58, 2005.

[4] E. Barka and R. Sandhu. Framework for role-based delegation models. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 168–176, 2000.

[5] E. Barka and R. Sandhu. A role-based delegation model and some extensions. In *23rd National Information Systems Security Conference*, pages 168–177, 2000.

[6] E. Barka and R. Sandhu. Role-based delegation model/hierarchical roles (rbdm1). In *Proceedings. 20th Annual Computer Security Applications Conference*, pages 396 – 404, 2004.

[7] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 20 – 30, 1990.

[8] P. P. Griggiths and B. W. Wade. An authorization mechanism for a relational database system. *ACM Trans. Database systems*, pages 242–55, 1976.

[9] C. Ionita and S. Osborn. Privilege administration for the role graph model. In Gudes and Shenoi, editors, *Research Directions in Data and Application Security*, pages 15–25. Kluwer, 2002.

[10] M. Nyanchama and S. Osborn. Access rights administration in role-based security systems. In J.Biskup, M. Morgenstern, and C. Landwehr, editors, *Database Security, VIII Status and Prospects*, pages 37–56. North-Holland, 1994.

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

[12] S. Osborn and Y. Guo. Modeling users in role-based access control. In *Fifth ACM Workshop on Role-Based Access Control*, 2000.

[13] W. Tolone, G.-J. Ahn, T. Pai, and S.-P. Hong. Access control in collaborative systems. *ACM Comput. Surv.*, 37(1):29–41, 2005.

[14] J. Wainer and A. Kumar. A fine-grained, controllable, user-to-user delegation method in rbac. In *Proceedings tenth ACM SACMAT*, pages 59–66, 2005.

[15] H. Wang and S. Osborn. An administrative model for role graphs. In *Data and Applications Security XVII*, pages 39–44. Kluwer, 2003.

[16] C. Wood and E. Fernandez. Decentralized authorization in a database system. In *Proceedings of the Fifth VLDB Conference*, pages 352–9, 1979.

[17] G. Yin, M. Teng, H.-M. Wang, Y. Jia, and D. xi Shi. An authorization framework based on constrained delegation. In *Proceedings of Parallel and Distributed Processing and Applications. Lecture Notes in Computer Science 3358)*, pages 845 – 57, Hong Kong, China, 2004.

[18] L. Zhang, G.-J. Ahn, and B.-T. Chu. A rule-based framework for role based delegation. In *Proceedings sixth ACM SACMAT*, pages 153–162, 2001.

[19] L. Zhang, G.-J. Ahn, and B.-T. Chu. A role-based delegation framework for healthcare information systems. In *Proceedings seventh ACM SACMAT*, pages 125–134, 2002.

[20] L. Zhang, G.-J. Ahn, and B.-T. Chu. A rule-based framework for role-based delegation and revokation. *ACM Trans. on Information and System Security*, 6(3):404 – 441, August 2003.

[21] X. Zhang, S. Oh, and R. Sandhu. PBDM: a flexible delegation model in RBAC. In *Proceedings of eighth ACM SACMAT*, pages 149–157, 2003.