Noname manuscript No.
(will be inserted by the editor)

Delegation in Role-Based Access Control

Jason Crampton · Hemanth Khambhammettu

the date of receipt and acceptance should be inserted later

Abstract User delegation is a mechanism for assigning access rights available to one user to another user. A delegation can either be a *grant* or *transfer* operation. Existing work on delegation in the context of role-based access control models has extensively studied grant delegations, but transfer delegations have largely been ignored. This is largely because enforcing transfer delegation policies is more complex than grant delegation policies. This paper, primarily, studies transfer delegations for role-based access control models. We also include grant delegations in our model for completeness. We present various mechanisms that authorize delegations in our model. In particular, we show that the use of administrative scope for authorizing delegations is more efficient than using relations. We also discuss the enforcement and revocation of delegations. Finally, we study delegation in the context of workflow systems. In particular, we demonstrate the application of the administrative scope and administrative domain concepts to control delegation of tasks in worklist-based workflow systems.

Keywords Authorization · Delegation · Role-based access control · Transfer · Grant · Workflow

1 Introduction

Role-based access control (RBAC) is being increasingly recognized as an efficient access control mechanism that facilitates security administration [1]. *Roles* are identified with various job functions in an organization and users are assigned to roles based on their job responsibilities and qualifications. Permissions are associated with roles. Users acquire permissions through the roles allocated for them. This feature of role-based models greatly simplifies the management of permissions.

Delegation is a mechanism of assigning access rights to a user. Delegation may occur in two forms: *administrative* delegation and *user* delegation. An administrative delegation allows an administrative user to assign access rights to a user and does not, necessarily, require that the administrative user possesses the ability to use the access right. A user delegation allows a user to assign a subset of his available rights to

Information Security Group, Royal Holloway, University of London

another user. However, a user delegation operation requires that the user performing the delegation must posses the ability to use the access right. Furthermore, like Schaad, we believe that an administrative delegation operation is often long-lived and more durable (permanent) than a user delegation operation that is short-lived (temporary) and intended for a specific purpose [2, Chapter 7, Page 117]. This paper studies user delegation. In the rest of the paper, 'delegation' is always to be understood as 'user delegation' unless stated otherwise. The user who performs a delegation is referred to as a 'delegator' and the user who receives a delegation is referred to as a 'delegatee'.

Rights can be delegated in two ways in RBAC: by delegating *roles* or by delegating individual *permissions*. Delegating a permission p gives the delegatee the ability to use p. However, delegating a role r gives the delegatee the ability to act in role r. That is, the delegatee is authorized for role r (and thereby gains the ability to use permissions assigned to role r and roles that are junior to r). In particular, we note that *individually* delegating all permissions explicitly assigned to a role r does not authorize the delegatee to act in role r.

Broadly, delegation of privileges may be classified into (at least) two kinds: *grant* and *transfer* [3]. A *grant delegation* model, following a successful delegation operation, allows a delegated access right to be available to both the delegator and delegatee. As such, this is a *monotonic* model, in which available authorizations are only increased due to successful delegation operations. However, in *transfer delegation* models, following a successful delegation operation, the ability to use a delegated access right is transferred to the delegatee; in particular, the delegated access right is no longer available to the delegator.

Grant delegation models are, primarily, concerned with allowing the delegatee to use the delegated access right. However, in transfer delegation models, besides allowing the delegatee to use the delegated access right, we must prevent the use of the delegated access right by the delegator. It is this feature that makes it more difficult to enforce transfer delegation policies in most access control frameworks [2, 4]. Furthermore, it can be easily seen that, in grant delegation models the availability of access rights increases monotonically with delegations. While some business requirements may support grant delegations, it is often desirable that sensitive access rights may not be available to a large number of users (at any given time). Such requirements are usually expressed as cardinality constraints in an access control policy [5, 6]. Transfer delegation policies prove to be more useful when an access control policy specifies cardinality limits on the availability of access rights between users.

Consider, for example, an access control policy that requires the co-operation of k users to accomplish a given task. The unavailability of (at least) one of the users, which can be for several reasons, makes it impossible to complete the task. A desirable solution would be for users to be able to delegate the access right to another user, who may act on behalf of the former user. In the above example, when assumed that the access control policy specifies that the right r, that is required to complete the task, is available to no more than k users, the reference monitor will always deny an attempt to delegate the right r, using *grant delegation*, to another user to prevent violation of the policy. Such scenarios require that the delegation be made using *transfer delegation*.

Most works that studied delegation in the context of role-based models are grant delegation models [3, 7–13]. To our knowledge no work has studied temporary transfer delegation for role-based models. This paper, primarily, aims to study transfer delegation for role-based models. Role hierarchies are an important reason for the wide interest in role-based models. Hence, it is natural to consider role hierarchies when studying

any aspect of role-based models. The semantics of transfer delegations become more complex when role-hierarchies are considered. We develop a comprehensive delegation model for role-based systems that provides support for both grant and transfer delegation policies. In this paper, we study delegation in the context of both $RBAC_0$ model (flat roles) and $RBAC_1$ model (hierarchical roles) of the RBAC96 family of models [1].

We also conduct a case study of our delegation model in the context of workflow systems. Most importantly, we demonstrate the application of the 'administrative scope' and 'administrative domain' concepts to authorize delegation operations in worklist-based workflow systems.

The rest of the paper is organized as follows. In the next section, we develop the background for the rest of the paper and define semantics for grant and transfer delegations. Section 3 describes the mechanism for authorizing delegations. Section 4 presents enforcement of delegations. A comparison of our work with related work in the literature is given in Sect. 5. We study delegation in the context of workflow systems in Sect. 6. We conclude this work and discuss future work in Sect. 7.

2 Role-Based Delegation

The theoretical development of role-based access control and its standardization has been strongly influenced by the RBAC96 family of models [1]. For this reason, we consider delegation within the context of RBAC96. In the next section, we introduce some important prerequisite concepts, including the relevant features of RBAC96 and the concept of administrative scope, which is the building block of the RHA family of administrative models [14].

2.1 Preliminaries

2.1.1 RBAC96.

$RBAC_0$ is the simplest model and defines a set of roles R, a set of permissions P, a set of users U, a permission-role assignment relation $PA \subseteq P \times R$, and a user-role assignment relation $UA \subseteq U \times R$. We denote the set of roles explicitly assigned to a user u by R_u; that is, $R_u = \{r \in R : (u, r) \in UA\}$ and the set of roles explicitly assigned to a permission p by R_p; that is, $R_p = \{r \in R : (p, r) \in PA\}$.

$RBAC_1$ introduces the concept of a *role hierarchy* $RH \subseteq R \times R$. The graph of the relation RH is acyclic and the transitive reflexive closure of RH defines a partial order on R. Role hierarchies are represented pictorially by the (directed) graph (R, RH), although the edges are typically shown without arrows.

We write $r \leqslant r'$ in preference to $(r, r') \in RH^*$ (the transitive reflexive closure of RH). We may also write $r' \geqslant r$ whenever $r \leqslant r'$. We write $\downarrow r$ to denote the set $\{r' \in R : r' \leqslant r\}$ and $\downarrow R'$ to denote $\bigcup_{r' \in R'} \downarrow r'$. We write $\uparrow r$ to denote the set $\{r' \in R : r' \geqslant r\}$ and $\uparrow R'$ to denote $\bigcup_{r' \in R'} \uparrow r'$.

Access control decisions are granted within the context of a *user session*, which is determined by the set of roles that a user activates. This set of roles is a subset of the roles for which the user is authorized directly by the UA relation and indirectly by the role hierarchy. We denote a session for user u by $S_u \subseteq \downarrow R_u$. A user u is permitted to

invoke a permission p if there exists an activated role $r \in S_u$ and a role $r' \in R$ such that $(p, r') \in PA$ and $r' \leqslant r$.

2.1.2 Administrative scope.

While RBAC96 is widely regarded as the *de facto* standard for role-based access control, there is less consensus regarding role-based administration. There are three approaches that are regarded as being relatively mature and sophisticated [15]: the ARBAC97 model [16], which is the administrative counterpart of RBAC96; the RHA family of models [14]; and the role control center [15]. Since delegation can be regarded as "lightweight user-based administration", it is natural that ideas from role-based administration may be useful in models for delegation. Indeed, the `can_delegate` relation in RDM2000 is very similar in structure and meaning to the `can_assign` relation from ARBAC97 [12]. In this section, we introduce the concept of administrative scope from the RHA model and generalize its definition for use in our delegation model.

Definition 1 (Crampton and Loizou [14]) *Let $r \in R$ and define $\sigma(r) = \{s \leqslant r : \uparrow s \subseteq \downarrow r \cup \uparrow r\}$. We say that $\sigma(r)$ is the* administrative scope *of r.*

Informally, $s \in \sigma(r)$ means that all paths in the graph (R, RH) starting at s pass through r. In Fig. 2(a) on page 8, for example, $\sigma(b) = \{d\}$; $g \notin \sigma(b)$, because $e > g$ and e is not comparable to b. Administrative scope defines a sub-hierarchy forming a natural unit of administration for the role r.

The success of any administrative operation in the RHA model is determined by the inclusion (or otherwise) of any role parameters in the requesting role's administrative scope. Hence, a user assigned to role b could add a new role i with parent role d, for example, but could not add a new role k with parent role g.

Administrative scope has been shown to be a far more flexible approach to role-based administration than ARBAC97 [14]. As such, it is unsurprising that it turns out to have considerable advantages in role-based delegation. However, it will be necessary to compute the administrative scope of a role in different partially ordered sets, each of which is a sub-hierarchy of the role hierarchy. Hence, we extend our notion of administrative scope to include two parameters: a partially ordered set X and an element $x \in X$. We write $\sigma(x, X)$ to denote the administrative scope of x computed in partially ordered set X. In practice X will be a subset of R. We now discuss the two most important cases.

- If a user u is assigned to a set of roles R_u, this induces a *user view* $\downarrow R_u$ of the role hierarchy containing all the roles to which u is implicitly assigned (via the user-role and role hierarchy relations). This user view is important in defining which roles a user retains following the delegation of a role. In this case, when u delegates r, we compute $\sigma(r, \downarrow R_u)$ (see Proposition 3).
- Similarly, a set of activated roles $S_u \subseteq \downarrow R_u$ defines a *session view* $\downarrow S_u$, which is useful in defining the roles available to a user following certain types of transfer delegation. In this case, when u delegates r, we compute $\sigma(r, \downarrow S_u)$ (see Proposition 4).

2.1.3 Administrative domains.

We say $D \subseteq R$ is an *administrative domain*, with administrator r, if $D = \sigma(r)$ for some $r \in R$. We write D_R to denote the set of administrative domains in R. We have the following useful result.

Lemma 2 (Crampton [17]) *Let $a, b \in R$. Then*

$$\sigma(a) \cap \sigma(b) = \begin{cases} \sigma(a) & \text{if } a \in \sigma(b), \\ \sigma(b) & \text{if } b \in \sigma(a), \\ \emptyset & \text{otherwise.} \end{cases}$$

Lemma 2 shows that any pair of administrative domains are either nested or disjoint [17]. This means that the set D_R ordered by subset inclusion can be represented as a tree. An illustration is given in Figure 1, where Figure 1(a) depicts a role hierarchy, Figure 1(b) shows administrative domains, which are enclosed by broken lines, and Figure 1(c) gives a tree representation of administrative domains.

2.2 Delegation Operations

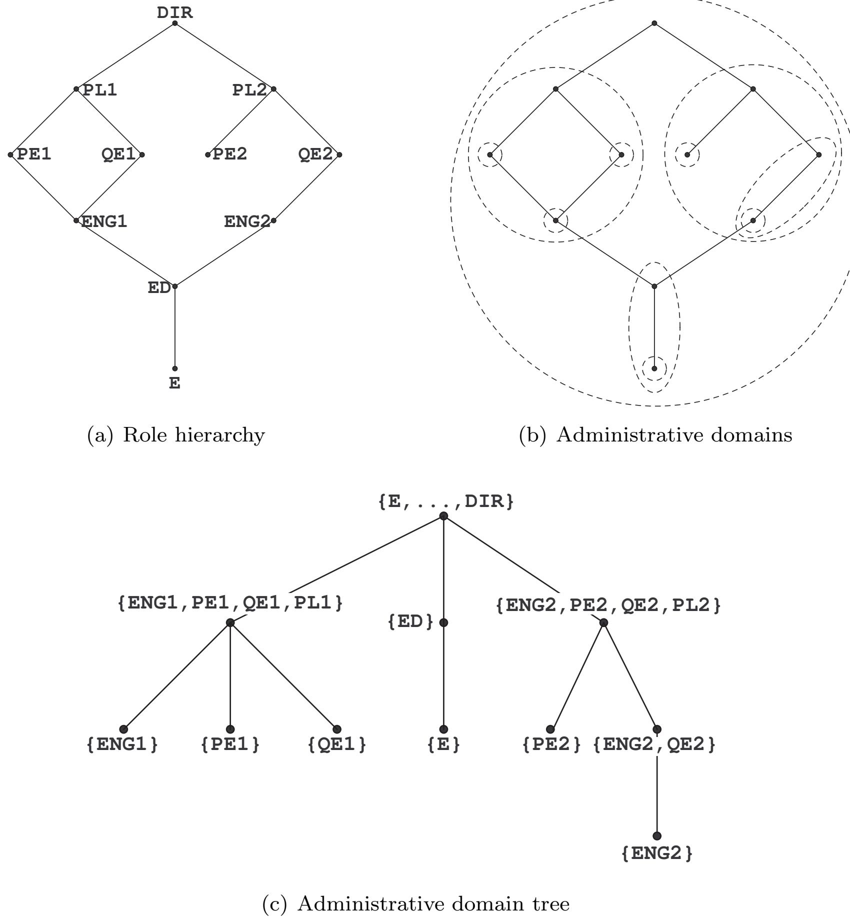
We now describe the characteristics of delegation operations in RBAC96. The grant operation is well understood and has been described in several earlier papers [3, 7–13]. We include it here for completeness. We define three different types of transfer operations in the context of RBAC_1.

2.2.1 $RBAC_0$.

`grantR`$_0(u, v, r)$: The delegator u grants the role r to delegatee v. The delegator may continue to use r.
`grantP`$_0(u, v, p)$: The delegator u grants the permission p to delegatee v. The delegator may continue to use p.
`xferR`$_0(u, v, r)$: The delegator u transfers the role r to delegatee v. The delegator may no longer use r.
`xferP`$_0(u, v, p)$: The delegator u transfers the permission p to delegatee v. The delegator may no longer use p.

2.2.2 $RBAC_1$.

`grantR`$_1(u, v, r)$: The delegator u grants the role r to delegatee v. The delegator may continue to use r. The delegatee acquires the right to use all roles in $\downarrow r$.
`grantP`$_1(u, v, p)$: The delegator u grants the permission p to delegatee v. The delegator may continue to use p.
`xferP`$_1(u, v, p)$: The delegator u transfers the permission p to delegatee v. The delegator may not continue to use p.
`xferRstrong`(u, v, r): The delegator u transfers the role r to delegatee v. The delegator may not use any role in $\downarrow r$. The delegatee acquires the right to use all roles in $\downarrow r$. We call this *strong transfer* of a role from the delegator to the delegatee.



(a) Role hierarchy

(b) Administrative domains

(c) Administrative domain tree

Fig. 1 An example role hierarchy

`xferRstatic`(u, v, r): The delegator u transfers the role r to delegatee v. The delegator may use $x \in \downarrow r$ *if there exist roles* $x = x_1, x_2, \ldots, x_k$ *such that* $x_i \neq r$, $x_i < x_{i+1}$ *and* $(u, x_k) \in UA$. (Informally, u retains the use of a role x if there is an alternative path from x to a role to which u is assigned.) We call this *static weak transfer* of a role from the delegator to the delegatee. As before, the delegatee acquires the right to use all roles in $\downarrow r$.

`xferRdynamic`(u, v, r): The delegator u transfers the role r to delegatee v. The roles available to the delegator u are determined by the roles that u activates in a session S_u. The delegator may use $x \in \downarrow r$ *if there exist roles* $x = x_1, x_2, \ldots, x_k$ *such that* $x_i \neq r$, $x_i < x_{i+1}$ *and* $x_i \in S_u$. (Informally, u regains the use of a role x if there is an alternative path from x to a role that u has activated in her session.) We call this *dynamic weak transfer* of a role from the delegator to the delegatee. As before, the delegatee acquires the right to use all roles in $\downarrow r$.

Given the above definitions, the following results are used as a basis for deciding whether the delegator is allowed to use a role following a successful transfer delegation operation. We discuss the enforcement of the consequences of transfer delegations in Sect. 4.

Proposition 3 *If u has performed a static weak delegation of role r, then u is denied access to all roles in $\sigma(r, \downarrow R_u)$, where R_u denotes the set of roles assigned to u.*

Proof Suppose u has performed a weak static delegation of role r. Let $x \in \sigma(r, \downarrow R_u)$. Then, by definition of administrative scope, $x \leqslant r$ and there does not exist a role $r' \in R$ such that $(u, r') \in UA$ and $x \leqslant r'$. The result follows.

Proposition 4 *If u has performed a dynamic weak delegation of role r, then u is denied access to all roles in $\sigma(r, \downarrow S_u)$, where S_u is the set of roles activated by u.*

Proof The proof is analogous to that of Proposition 3.

Proposition 5 *Let R_{strong}, $R_{dynamic}$ and R_{static} denote the set of roles available to a user following the operations* `xferRstrong`(u, v, r)*,* `xferRdynamic`(u, v, r) *and* `xferRstatic`(u, v, r)*, respectively. Then $R_{strong} \subseteq R_{dynamic} \subseteq R_{static}$.*

Proof $R_{strong} = \downarrow R_u \setminus \downarrow r$, $R_{dynamic} = \downarrow R_u \setminus \sigma(r, \downarrow S_u)$ and $R_{weak} = \downarrow R_u \setminus \sigma(r, \downarrow R_u)$. Now $\sigma(r) \subseteq \downarrow r$ irrespective of the sub-hierarchy in which σ is computed. Hence $R_{strong} \subseteq R_{dynamic}$. Moreover, $S_u \subseteq R_u$ and it is easy to see that this implies that $\sigma(r, \downarrow R_u) \subseteq \sigma(r, \downarrow S_u)$ (since if $r' \geqslant x$ for some $x \in \downarrow r$ and some $r' \in S_u$, then $r' \in R_u$).

Consider the role hierarchy depicted in Fig. 2. We assume that some user u is assigned to roles b and f. In the diagram, unfilled nodes represent roles that are available to u (and filled nodes represent nodes that are not available). Each diagram represents different views of the role hierarchy: Fig. 2(a) illustrates the whole hierarchy; the remaining figures illustrate the effect of different types of delegation.

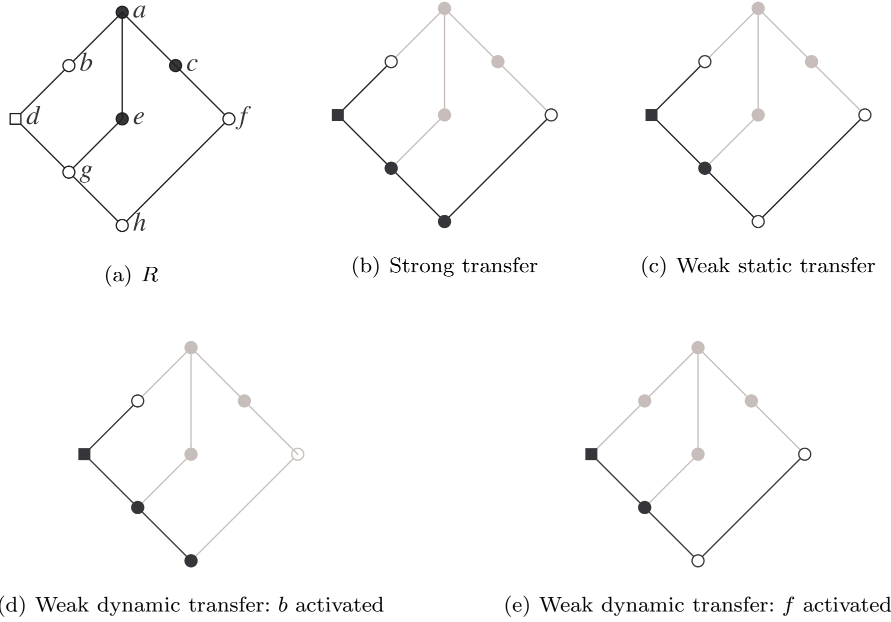
The user delegates role d, represented by the square node, to another user. Figure 2(b) illustrates the roles that are denied to the user in the event that strong transfer is used to delegate the role. If we are using static weak delegation, the role h is always available, as depicted in Fig. 2(c). If we are using dynamic weak transfer, then the ability to use role h is determined by whether the user activates role f, as shown in Fig. 2(d) and Fig. 2(e).

3 Controlling Delegation

We assume that an access control policy specifies whether delegation of a role or permission is permitted. We also assume the presence of a reference monitor that evaluates such access control policies to determine whether a delegation operation is permitted. There are two issues involved in the specification of delegations:

- Is a user (delegator) authorized to delegate a role or permission that is available to him?
- Can a role or permission be delegated to a user (delegatee)?

In this section, we describe two new mechanisms for authorizing delegations, and discuss their advantages over existing approaches.



(a) R (b) Strong transfer (c) Weak static transfer

(d) Weak dynamic transfer: b activated (e) Weak dynamic transfer: f activated

Fig. 2 Transfer delegation patterns for role d

3.1 Delegation Relations

3.1.1 Role delegation relation.

We introduce two relations that authorize role delegations in a role-based system: `can-delegate` and `can-receive`. The binary relation `can-delegate` specifies the set of roles that can be delegated by a delegator and the relation `can-receive` authorizes delegation of a role to a delegatee.

The relation `can-delegate` $\subseteq R \times R$ specifies whether a user is authorized to delegate a role: $(r, r') \in$ `can-delegate` specifies that a user u is authorized to delegate role r', if $r \in S_u$, where S_u denotes the set of roles that are active in u's current session(s). For example, in Fig. 2(a), if $(b, d) \in$ `can-delegate` then u may delegate role d if $b \in S_u$.

We define a constraint on the tuples in the `can-delegate` relation that guarantees that user u can not delegate a role r unless u is assigned to it. Specifically, we require that if $(r, s) \in$ `can-delegate` then $r \geqslant s$. In our example, in Fig. 2(a), an attempt to add the tuple (b, d) to the `can-delegate` relation will succeed since $b \geqslant d$. However, an attempt to add the tuple (d, c) to the `can-delegate` relation will fail since $d \ngeqslant c$.

A delegatee must satisfy a set of conditions, usually role memberships, in order to be authorized to receive a delegation. A *delegation condition* specifies the conditions that must be satisfied by the delegatee to receive a delegation. We adopt the notation of a SARBAC constraint to model a delegation condition [14]. Let $R' = \{r_1, \ldots, r_k\}$ be a subset of R and let $\bigwedge R'$ denote $r_1 \wedge \cdots \wedge r_k$. We model a delegation condition as $\bigwedge C$ for some $C \subseteq R$. A delegation condition $\bigwedge C$ is said to be *satisfied* by a user

v if $C \subseteq \downarrow R_v$. In other words, the condition $\bigwedge C$ is satisfied by the delegatee if he is assigned to all the roles in C. The relation `can-receive` $\subseteq R \times C$ specifies whether a user is authorized to receive a role delegation: $(r, C) \in$ `can-receive` means a delegatee v may receive a delegation of role r provided that v satisfies $\bigwedge C$.

It may be more efficient to limit the set of roles that a delegatee may receive due to a successful delegation. For example, in Fig. 2(a), is it reasonable for a delegation request to succeed that delegates a role c to a delegatee v who currently is assigned to role g? We may often require that such delegations are not allowed, since v may lack sufficient expertise to efficiently perform the job requirements of role c. Essentially, we require that a delegation always results in a gradual elevation of privileges for the delegatee with respect to the role hierarchy. We formalize the above requirement by defining the following constraint on the tuples in the `can-receive` relation: $(r, C) \in$ `can-receive` then for all $r' \in C$ we require that $r' < r$. Note that this constraint does not apply if r has no junior roles (that is, r is a leaf node in the hierarchy). In our example, $(c, \{g\})$ is not a permissible entry in `can-receive`; $(c, \{f\})$, in contrast, is a permissible entry.

In summary, the role-based reference monitor refers to the `can-delegate` and `can-receive` relations to establish whether a delegation operation can succeed. A request by u to delegate r to v will succeed only if there exists $(r', r) \in$ `can-delegate` such that $r' \in S_u$ and a tuple $(r, C) \in$ `can-receive` such that v satisfies $\bigwedge C$. In our example, let us assume that $(b, d) \in$ `can-delegate`, $(d, \{g\}) \in$ `can-receive`, $(u, b) \in UA$ and $(v, g) \in UA$. Then an attempt by u to delegate role d to v will succeed.

3.1.2 Permission delegation relation.

We define a relation `can-delegatep` $\subseteq R \times P$ that specifies the set of permissions that can be delegated by a user u: $(r, p) \in$ `can-delegatep` specifies that a delegator u may delegate a permission p provided that there exists $r \in R_s$. As for the `can-delegate` relation, we define a constraint on the `can-delegatep` relation that guarantees that no user u can delegate a permission p that is not available to u. In other words, we require that if $(r, p) \in$ `can-delegatep` then there exists a role r' such that $(p, r') \in PA$ and $r' \leqslant r$.

The relation `can-receivep` $\subseteq P \times C$ specifies whether a user is authorized to receive a permission delegation, where C is a delegation condition as defined above. $(p, C) \in$ `can-receivep` means that a delegatee v is authorized to receive a delegation of a permission p if v satisfies $\bigwedge C$. As for the `can-receive` relation, we define a constraint on the `can-receivep` relation that ensures that a delegation always results in a natural progression for the delegatee with respect to the role hierarchy. If $(p, C) \in$ `can-receivep` then there must exist $r' \in C$ and $r \in R$ such that $(p, r) \in PA$ and $r' < r$. As in the previous section, this constraint does not apply if p is only assigned to roles that are leaf nodes in the hierarchy.

3.2 Using Administrative Scope

The use of relations for controlling delegations, discussed above, has been used extensively in the literature mainly perhaps because of its simplicity [7, 12, 13]. The use of relations is simple, if we assume that RBAC relations, such as the role hierarchy

relation RH, remain static. However, updates to the RBAC relations may lead to inconsistencies in the `can-delegate` and `can-receive` relations. Such inconsistencies are explained in detail in Sect. 3.3. For now, we note that the dynamic nature of various RBAC components increases the complexity of managing the relations that are used for controlling delegations. It is our belief that the mechanism used for controlling delegations must be simple and able to implicitly handle any updates to RBAC relations. In this section, we suggest an alternative method for controlling delegations, which dynamically handles updates to RBAC relations using the concept of administrative scope [14].

We limit the extent to which a user can delegate roles and permissions using the administrative scope of the roles(s) activated by the user. Specifically, we define the administrative scope of a session s to be

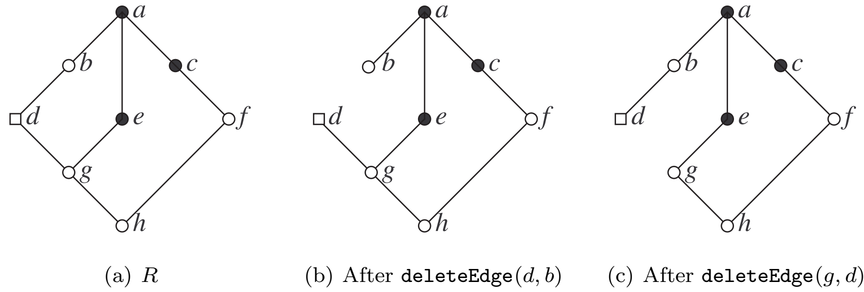
$$\sigma(s) = \bigcup_{r \in s} \sigma(r).$$

Then in order for the delegation of role r by user u to succeed we require that $r \in \sigma(s)$, where s is u's current session. In other words, u can only delegate roles that are within his administrative scope. Similarly, in order for the delegation of permission p by user u to succeed we require that there exists $r \in s$ such that $(p, r') \in PA, r' \leqslant r$ and $r' \in \sigma(s)$.

We now consider what criteria the delegatee must satisfy to be able to receive a delegation. Informally, for the delegation of r to v to succeed, we require that v is already "sufficiently authorized". Now, the delegation of role r means that the delegatee is authorized for all roles $r' < r$. These observations lead to the idea that the delegatee should already be assigned to any roles outside the delegator's administrative scope that the delegatee will acquire as a result of the delegation. More formally, for the delegation of a role r to user v by user u to succeed we require that for all $r' < r$ such that $r' \notin \sigma(s)$, there exists r'' such that $r' \leqslant r''$ and $(v, r'') \in UA$.

In our example, Fig. 2(a), u (who is assigned to b and f) may delegate role d since $d \in \sigma(b)$. Moreover, the delegation of d to user v will only succeed if v is already assigned to role g. Note, however, that a user w who is assigned (only) to role f will not be able to receive the delegated role d (from u) because $g \notin \sigma(s)$, $g < d$ and w is not assigned to g.

Administrative scope implicitly deals with any updates to various RBAC relations, in particular the role hierarchy relation RH, since administrative scope of a session s is computed, dynamically, with respect to the role hierarchy. Consider, for example, Fig. 3(a), which depicts our original role hierarchy, and Fig. 3(b), which depicts the role hierarchy that is obtained after deleting an edge between roles d and b. In the original hierarchy, if a user u has activated b in a session s then u is authorized to delegate role d since $d \in \sigma(s)$. However, following the edge deletion operation, u is no longer able to delegate d since $d \notin \sigma(s)$. In other words, the success of delegation operations adapts in a natural and transparent way to changes in the structure of the role hierarchy. Figure 3(c) depicts the role hierarchy obtained after deleting an edge between roles g and d. In the original hierarchy, the delegatee w, who is assigned to role f, can not receive a delegation of role d since $g < d$ and w is not assigned to g. Following the edge deletion operation, however, w will be able to receive a delegation of d because there are no roles less than d. More generally, using this approach (that is, based on administrative scope), any role that is a leaf node in the hierarchy may be delegated to any user.



(a) R (b) After `deleteEdge`(d, b) (c) After `deleteEdge`(g, d)

Fig. 3 Role hierarchies *before* and *after* edge deletion

3.3 Discussion

Most work in the literature uses a single relation to control delegation [7, 12, 13]. This relation encodes conditions on both the delegator and delegatee. We have proposed an alternative approach where the above issues are dealt with independently. There are several advantages in dealing with these issues separately. Primarily, it eases the management of delegation specification. Furthermore, such an approach employs separation of tasks, thus, making the model less error prone while updating delegation policies. For example, if we wish to revoke the authority of a role r to perform any delegations, then appropriate tuples are deleted from the `can-delegate` relation. Note that such an operation does not require any updates to the `can-receive` relation that authorizes a delegation to the delegatee. Similarly, if we require that a permission p may no longer be delegated to a delegatee who satisfies a condition $\bigwedge C$, we only delete necessary tuples from the `can-receivep` relation.

The use of relations to control delegation is common to most existing approaches [7, 12, 13]. The use of relations is appropriate, if we assume that RBAC structures such as the role hierarchy remain static. However, updates to the role hierarchy may lead to inconsistencies in the `can-delegate` and `can-receive` relations. Consider, for example, Fig. 3(a) and Fig. 3(b) that depict an original role hierarchy and a role hierarchy obtained after deleting an edge between roles d and b. If $(b, d) \in$ `can-delegate` it will need to be *explicitly* deleted following the deletion of this edge. Similar considerations apply to the operations that involve revocation of a permission p assigned to b. Similarly, if $(b, \{d\}) \in$ `can-receive`, then we must delete this entry following the deletion of the edge between b and d. In summary, the `can-delegate` and `can-receive` relations must be updated after updates to certain RBAC relations to prevent inconsistencies.

Hence the advantages of using relations for controlling delegations are limited if we allow updates to RBAC relations. In contrast, administrative scope is a *dynamic* model and *implicitly* deals with any updates to RBAC relations. It is important to note that, following successful updates to RBAC relations, no explicit updates are required with the use of administrative scope to resolve any inconsistencies involved in controlling delegations. This is because administrative scope (and hence delegation) is determined by the structure of the role hierarchy. In short, the use of administrative scope greatly simplifies delegation in the presence of dynamic RBAC structures.

Furthermore, the administrative scope model for controlling delegations preserves the separation of conditions on delegator and delegatee that we introduced in our relation-based model. The issues that are involved in controlling delegations, such as specifying the roles that a user is authorized to delegate and receive, are still dealt with independently. Note also that the constraint we use to limit the increase of power of the delegatee can be framed very elegantly using administrative scope and existing RBAC relations.

4 Enforcing Delegation Constraints

Enforcing delegation operations that grant a role or permission to a user is quite straightforward as it is a monotonic action. That is, the set of roles or permissions for each user either stays the same or increases. Delegation operations that transfer a role or permission are rather more difficult. In this section, we introduce three relations that are used to guarantee that delegation operations are enforced correctly. The model described in this section can be used for both grant and transfer delegations. To our knowledge, this is the first treatment of the consequences of temporary transfer delegation policies for access control in role-based systems. This is mainly due to the fact that grant delegations are easy to enforce, whereas transfer delegations are difficult to enforce because of their non-monotonicity.

4.1 The Delegation History Relation

The delegation history (DH) relation is used to record all delegations that have been made. The DH relation is used by the delegators and administrative users for administrative purposes. Typically, such activities include auditing and revoking delegations.

A DH tuple has the form (i, u, v, o, C, M) where i is an identifier, u is the delegator, v is the delegatee, o is the object or target of the delegation that can either be a set of roles $R' \subseteq R$ or a set of permissions $P' \subseteq P$, C is the set of conditions, from the `can-receive` relation, that must be satisfied by the delegatee and M is a *delegation mask*. A delegation mask records various internal details of a delegation.

A delegation mask M contains five bits which are used to record certain details of the delegation. More specifically, a delegation mask has the form $b_4b_3b_2b_1b_0$, where $b_i \in \{0, 1\}$, b_4 specifies whether the delegated object can be further delegated[1], b_3 specifies whether the delegated object is a role or permission, b_2 specifies whether the delegation is static or dynamic, b_1 specifies whether the delegation is strong or weak, and b_0 specifies whether the delegation is grant or transfer. The characteristics of a delegation mask are summarized in Table 1, and some examples of masks are shown in Table 2 together with the commands that would give rise to such a mask.

Note that we require the delegator to explicitly set some of the bits in the delegation mask while performing a delegation operation. In particular, we require that the bits b_0, b_1 and b_2 are set by the delegator, while the bits b_3 and b_4 are deduced by the access control enforcement system.

[1] An extra boolean-valued parameter can be added to each of the delegation commands; this parameter would be used to set this bit.

Table 1 Delegation mask bit values

	b_4	b_3	b_2	b_1	b_0
0	`undelegatable`	`role`	`static`	`strong`	`grant`
1	`delegatable`	`permission`	`dynamic`	`weak`	`transfer`

Table 2 Examples of valid delegation mask values, their semantics, and associated commands

Mask	**Delegation semantics**	**Command**
`00xx0`	undelegatable, role, grant	`grantR`
`10x01`	delegatable, role, strong, transfer	`xferRstrong`
`00011`	undelegatable, role, static, weak, transfer	`xferRstatic`
`10111`	delegatable, role, dynamic, weak, transfer	`xferRdynamic`
`11xx0`	delegatable, permission, grant	`grantP`
`11x01`	delegatable, permission, strong, transfer	`xferPstrong`
`01111`	undelegatable, permission, dynamic, weak, transfer	`xferPdynamic`
`11011`	delegatable, permission, static, weak, transfer	`xferPstatic`

4.2 Temporary Delegation Relations

We also introduce two relations *tempUA* and *tempPA*, which record temporary user-role and user-permission assignments that arise from delegation operations: *tempUA* contains tuples of the form (i, u, r, s), where $s \in \{-, +\}$ and i identifies a tuple in the DH relation. The meaning of the tuple $(i, v, r, +)$ is analogous to $(v, r) \in UA$; such a tuple would arise as a result of a grant or transfer of role r to the delegatee v. In contrast, a tuple of the form $(i, u, r, -)$ means that u is prohibited from activating role r; such a tuple only arises when u transfers r to v. The precise set of roles that are unavailable for the delegator u depends on the transfer type used for delegation (strong, weak static or weak dynamic).

The *tempPA* relation contains tuples of the form (i, u, p, s), where $s \in \{-, +\}$ and i identifies a tuple in the DH relation. The tuple $(i, v, p, +)$ means that v is allowed to use p; such a tuple would arise as a result of a grant or transfer of permission p to the delegatee v. A tuple of the form $(i, u, p, -)$ means that u is prohibited from invoking permission p; such a tuple only arises when u transfers p to v.

For simplicity, we write (i, u, R', s) to denote $\{(i, u, r', s) : r' \in R'\}$. We also write (i, u, P', s) to denote $\{(i, u, p', s) : p' \in P'\}$. We now describe the effects on various relations following the successful execution of a delegation operation. We focus our attention on the more difficult case of $RBAC_1$.

Successful delegation operations are recorded in the delegation history relation DH. Hence, following any delegation of o by u to v, we have $DH \leftarrow DH \cup \{(i, u, v, o, C, M)\}$. In addition we have the following operational semantics:

$\texttt{grantR}_1(u, v, r)$: u grants role r to v. Such a delegation requires that the delegatee v is allowed to use role r. Hence, $tempUA \leftarrow tempUA \cup \{(i, v, r, +)\}$.

$\texttt{grantP}_1(u, v, p)$: u grants permission p to the delegatee v. Then, permission p must be available to the delegatee v. Hence, $tempPA \leftarrow tempPA \cup \{(i, v, p, +)\}$.

$\texttt{xferP}_1(u, v, p)$: delegator u transfers the authority to use permission p to the delegatee v. Hence, $tempPA \leftarrow tempPA \cup \{(i, v, p, +), (i, u, p, -)\}$.

`xferRstrong`(u, v, r): u performs a strong transfer of role r to v. The delegator u may not use any role in $\downarrow r$ and the delegatee v acquires the right to use all roles in $\downarrow r$. Hence, $tempUA \leftarrow tempUA \cup \{(i, v, r, +), (i, u, R', -)\}$, where $R' = \downarrow r$.

`xferRstatic`(u, v, r): u performs a static weak transfer of role r to v. The delegator u may not use a role $x \in \downarrow r$ unless there exists a role r' such that $(u, r') \in UA$ and $x \leqslant r'$. Hence, by Proposition 3, $tempUA \leftarrow tempUA \cup \{(i, v, r, +), (i, u, R', -)\}$, where $R' = \sigma(r, \downarrow R_u)$.

`xferRdynamic`(u, v, r): u performs a dynamic weak transfer of role r to v. The set of roles available to the delegator u are computed by the roles that u activates in a session. Hence, by Proposition 4, $tempUA \leftarrow tempUA \cup \{(i, v, r, +), (i, u, R', -)\}$, where $R' = \sigma(r, \downarrow S_u)$.[2]

4.3 Access Control Decisions

The role-based reference monitor uses the *tempUA* and *tempPA* relations, in addition to the *UA* and *PA* relations, to make access control decisions. For example, an attempt by a user u to activate a role r is always denied if there exists a tuple $(i, u, R', -) \in tempUA$ such that $r \in R'$; and granted if there exists a tuple $(i, u, r', +) \in tempUA$ or $(u, r') \in UA$ such that $r \leqslant r'$. Similarly, an attempt to invoke a permission p by u is always denied if $(i, u, p, -) \in tempPA$; and granted if $(i, u, p, +) \in tempPA$ or there exist $r, r' \in R$ such that $(p, r) \in PA$, $(u, r') \in UA$ and $r \leqslant r'$.

4.4 Revocation

Successful delegations have a specified lifetime or may be revoked before the delegation ends.[3] In either case, the *tempUA* and *tempPA* relations must be updated to prevent any subsequent use of the delegated access right by the delegatee and, if necessary, to allow the delegator to use the delegated access right. Essentially, we require that when a delegation $d \in DH$ ends or is revoked the tuples $(i, u, R', s) \in tempUA$ and $(i, u, P', s) \in tempPA$ are *deleted*, where i identifies the obsolete delegation d. We now describe the effects on various relations when a delegation ends or is revoked.

$\texttt{grantR}_1(u, v, r)$: When such a delegation ends, the delegatee v is no longer allowed to use role r. Hence, $tempUA \leftarrow tempUA \setminus \{(i, v, r, +)\}$.

$\texttt{grantP}_1(u, v, p)$: permission p must no longer be available for the delegatee v. Hence, $tempPA \leftarrow tempPA \setminus \{(i, v, p, +)\}$.

$\texttt{xferP}_1(u, v, p)$: delegatee v must be prevented from using permission p and the delegator u regains the ability to use the delegated permission p. Hence, $tempPA \leftarrow tempPA \setminus \{(i, v, p, +), (i, u, p, -)\}$.

[2] It is important to note that, following a dynamic weak transfer, the roles that are to be prevented for the delegator u are determined by the roles that are active in u's current session. Hence, it is necessary to add/delete appropriate tuples to the *tempUA* relation whenever a session is initiated or updated by a user who has performed a weak dynamic delegation.

[3] In a practical implementation, the *DH* relation would include some information that would determine the lifetime of the delegation: this information might be a start and an end time, or a start time and a duration, for example.

`xferRstrong`(u, v, r): u performs a strong transfer of role r to v. The delegator u gets back the right to use any role in $\downarrow r$ and the delegatee v can no longer use any role in $\downarrow r$. Hence, $tempUA \leftarrow tempUA \setminus \{(i, v, r, +), (i, u, R', -)\}$ where $R' = \downarrow r$.

`xferRstatic`(u, v, r): u performs a static weak transfer of role r to v. Similar to the above delegation, the delegator u gets back the right to use any role in $\downarrow r$ and the delegatee v can no longer use any role in $\downarrow r$. Hence, $tempUA \leftarrow tempUA \setminus \{(i, v, r, +), (i, u, R', -)\}$ where $R' = \sigma(r, \downarrow R_u)$.

`xferRdynamic`(u, v, r): u performs a dynamic weak transfer of role r to v. Again, the delegator u regains the right to use any role in $\downarrow r$ and the delegatee v can no longer use any role in $\downarrow r$. Hence, $tempUA \leftarrow tempUA \setminus \{(i, v, r, +), (i, u, R', -)\}$ where $R' = \sigma(r, \downarrow S_u)$.[4]

5 Related Work

Several delegation models have been proposed for role-based access control [3, 7–13]. The early work of Barka and Sandhu was instrumental in identifying the important considerations for delegation in RBAC [3]. This included the concepts of *monotonic* and *non-monotonic* delegation, which correspond to our grant and transfer models. To our knowledge, the work that we present in this paper represents the first attempt to deal properly with the consequences of temporary non-monotonic delegation in RBAC.[5]

Of the work in the literature, the RBDM0, RDM2000 and PBDM models are closest to our work [7, 8, 12, 13]. RBDM0 is a model for delegating roles, and is based on the RBAC_0 model of the RBAC96 family of models [7]. Another role delegation model for role-based access control is presented in [8]. RDM2000 defines a rule-based framework for delegation and revocation [12]. The model considers role hierarchies and also provides support for multi-step delegations. The PBDM model proposes a delegation model for permissions that supports multi-step delegations [13].

The RDM2000 model uses a relation $can_delegate \subseteq R \times 2^R \times \mathbb{N}$ to authorize delegations, where $\mathbb{N}$ is the set of natural numbers. If $(r, C, n) \in can_delegate$, a delegator acting in role r may delegate any role $r' \leqslant r$ to a delegatee v provided v satisfies some role assignment conditions specified by C; n is used to define the maximum depth of a delegation. A major limitation of this relation is that no constraints are defined on the tuples in the *can_delegate* relation. That is, r and C are not required to have any relationship with each other. What this means is that the delegatee's power can be arbitrarily increased by a successful delegation. In practice, a delegatee can be assigned to roles for which they lack any relevant expertise or experience.

The PBDM model uses a similarly named relation $can_delegate \subseteq R \times R \times P \times \mathbb{N}$. If $(r, r', P', n) \in can_delegate$ then a delegator who is assigned to role r can delegate the set of permissions P' to a user who is assigned to a role r' with a maximum delegation depth of n. Like the RDM2000 model, the *can_delegate* relation does not

[4] Recall that, following a dynamic weak transfer of a role r, we determine the roles that are not available for the delegator u, in a session s, by computing $\sigma(r, \downarrow S_u)$ (see Proposition 4). Hence, it may also be necessary to delete appropriate tuples from the *tempUA* and *tempPA* relations when the delegator u deactivates a role.

[5] Barka distinguishes between temporary and permanent delegation for role-based delegation models [18]. We believe the latter is more correctly viewed as an administrative activity. Barka does not consider temporary non-monotonic delegation policies (which we call transfer delegation policies).

Table 3 Delegation characteristics in various delegation models

Characteristic	RBDM0	RDM2000	PBDM	Our model
Role delegation	✓	✓	✓	✓
Permission delegation	✗	✗	✓	✓
Grant delegation	✓	✓	✓	✓
Transfer delegation	✗	✗	✗	✓
Controlling delegations	✓	✓	✓	✓
Implicit updates	✗	✗	✗	✓
Delegation history	✗	✓	✗	✓
Temporary delegation assignments	✓	✓	✓	✓
Revocation	✓	✓	✓	✓

require that there is any relationship between P' and r, which means that it is possible for a delegator u to delegate a permission p that is not available to u. As we have already noted, these relation-based approaches to delegation are unlikely to be suitable in environments where the role hierarchy may change.

Unlike the models discussed above, which use a single relation for controlling delegations, we use two different relations: `can-delegate` and `can-receive`. The advantages of using different relations for controlling delegations include flexibility, greater control while specifying delegations, ease of management and is less error prone. Furthermore, our model for controlling delegations defines constraints on the `can-delegate` and `can-receive` relations. Such constraints ensure that the tuples that are added to the `can-delegate` and `can-receive` relations does not give the authority for a delegator u to delegate a right that is not available to u and the rights of a delegatee v can only be incrementally increased and are limited by v's existing rights.

Hence, we believe our delegation specification model is more conservative (and thus safer), more fine-grained and more manageable than these models [7, 8, 12, 13]. However, in Sect. 3.3, we observed that the use of relations for controlling delegations may not be efficient for implicitly handling updates to various RBAC relations. Our model includes an alternative way of controlling delegations using the concept of administrative scope [14]. The administrative scope model is dynamic and implicitly handles any updates to RBAC relations, in particular the role hierarchy relation RH. We have also described the enforcement of both grant and transfer delegations and dealt with revocations.

A delegation model with restricted permission inheritance is proposed in [9]. The model is based on the idea of dividing a role hierarchy into inter-related role hierarchies. A cascaded role delegation model in the context of decentralized trust management systems is presented in [10]. Another model that supports delegation of both roles and permissions is discussed in [11]. We believe that our model can easily be extended to incorporate these concepts. However, none of the above work considers transfer delegation for role-based systems. Table 3 compares the support provided for key delegation features by well known delegation models.

We have considered revocation scenarios where a delegation is either revoked by the delegator or has expired. However, more complex revocation policies may be considered. Most importantly, we may also consider revocation of a delegation when a delegatee user is revoked from a role $r \in C$, where C is a delegation condition. In other words, since the delegatee no longer satisfies at least one of the conditions that led to the suc-

cess of the delegation, we require such a delegation to be revoked. Hagström *et al* have classified revocation into three dimensions: resilience, propagation and dominance [19]. *Resilience* distinguishes an explicit revocation of a (delegated) positive authorization for privilege α with issuance of a negative authorization for the privilege α.[6] *Propagation* distinguishes the effects of a revocation with respect to space. If a delegation is revoked and this delegation had been used to generate other delegations, then should the revocation be applied only to the initial delegation (local/non-cascading revocation) or must subsequent delegations also be revoked (global/cascading revocation)? *Dominance* deals with revocation scenarios where the delegatee user v has gained an authorization α from more than one delegator. If u delegates α to both u' and v, and u' also delegates α to v then u may choose between revoking only the direct delegation of α from u to v or all delegations of α that originate from u.

The enforcement of complex revocation policies, such as those discussed above, will still have the same effect, as described in Section 4.4, on the *tempUA* and *tempPA* relations. Suppose, for example, a user u had issued a delegation $\texttt{grantP}_1(u, v, p)$. The success of such a delegation yields $\mathit{tempPA} \leftarrow \mathit{tempPA} \cup \{(i_k, v, p, +)\}$. Let us assume that u had initiated a revocation request for the above delegation issued to delegatee v. If the revocation is local (non-cascading), then the effect of such a revocation is as described earlier. That is, $\mathit{tempPA} \leftarrow \mathit{tempPA} \setminus \{(i_k, v, p, +)\}$. If the revocation is global (cascading) then we have two cases: a user may issue no more than one simultaneous delegation of authorization α and a user may issue more than one simultaneous delegation of α.[7] If we limit a user to issue no more than one simultaneous delegation of α then the set of delegation identifiers $\{i_1, \ldots, i_n\} \in I$, where $i_k, 1 \leqslant k \leqslant n$, identifies a unique delegation $d \in DH$, can be represented as a linear list L. Alternatively, if we allow a delegator to issue more than one simultaneous delegation of α then the set of identifiers $\{i_1, \ldots, i_n\} \in I$ can be represented as a tree T. Hence, in the above example, while enforcing a cascading revocation with the former case, we need to delete delegations of permission p with identifier $\{i_k, \ldots, i_n\} \in L$ from the *tempPA* relation. In the latter case, we need to delete all delegations of permission p rooted at $i_k \in T$ from the *tempPA* relation. It is important to note that the enforcement of a 'transfer delegation' revocation is always cascading. Hence, we believe that more complex revocation policies can easily be incorporated in the revocation mechanism that has been described in this paper.

6 Delegation in Workflows

A workflow identifies various activities that are involved in an organizational or a business process. A computerized workflow system automates such processes. In a computerized workflow, typically, *activities* that are part of a process are represented as *tasks*.

A workflow may be executed in (at least) two ways: either tasks that are to be performed may be assigned (forwarded) to users or users may initiate requests to perform tasks. The Workflow Management Coalition (WfMC) recommends a "workflow reference model", in which tasks that are to be performed are assigned to users [20]. Subsequently, the user performs the task that has been assigned. There exist several

---

[6] Note that we do not consider negative authorizations in our model.

[7] Note that we have to deal with cascading revocation only when we allow multi-step delegations.

commercial products that conform to the WfMC reference model [21, 22]. For the purposes of our discussion, we will use this workflow execution model.

A workflow specification is a partially ordered set of "abstract" tasks T. A workflow management system instantiates a workflow specification, which is referred to as an *instance* of the workflow, in order to execute a workflow specification. If $\mathsf{t} < \mathsf{t}'$ then t must be performed before t' in any instance of the workflow. We will write t to denote an instance of the abstract task t in the workflow specification. Each instance of a workflow is associated with a *tasklist*, which is a set of task-user assignment pairs, with the assumption that the user to whom a task is assigned, performs the task.[8] A tasklist ω is represented as a set of task-user pairs $[(t_1, u_1), \ldots, (t_n, u_n)]$, it is assumed that u_i is authorized to perform task t_i where $1 \leqslant i \leqslant n$.[9]

A *workflow authorization model* authorizes users to perform tasks in a workflow. Recent research has considered the use of role-based access control as an authorization model for workflows [23, 24]. A role-based workflow authorization model defines a set of roles R, a set of tasks T, a set of users U, a task-role assignment relation $TA \subseteq \mathsf{T} \times R$, a user-role assignment relation $UA \subseteq U \times R$ and a role hierarchy relation $RH \subseteq R \times R$. Users are authorized to perform tasks for which they are authorized based on the roles that are assigned to them. For example, a user $u \in U$ is authorized to perform a task $\mathsf{t} \in \mathsf{T}$ if there exist $r, r' \in R$ such that $(\mathsf{t}, r) \in TA$, $r \leqslant r'$ and $(u, r') \in UA$. For simplicity, we will write $(\mathsf{t}, u) \in \mathcal{A}$ if u is authorized to perform t.

User delegation, in traditional access control, is a mechanism that permits a user to assign a subset of her assigned authorizations to other users who currently do not possess the authorization. However, in a tasklist-based workflow model, user delegation means that a user *re-assigns* a task that is assigned to her, in an instance of a workflow, to another user. In other words, delegation of a task instance is essentially an update to the tasklist ω, which has the effect of changing the associated authorized user. More formally, should a request by a user u to delegate a task t to a user v succeed then $\omega \leftarrow (\omega \cup \{(t, v)\}) \setminus \{(t, u)\}$. It is important to note here that delegation of a task (assignment) in tasklist-based workflows is analogous to a "permission transfer" delegation operation. In other words, in the given instance of the workflow, the delegatee acquires the right to perform the delegated task and the delegator is no longer authorized to perform the delegated task.

We now discuss the delegation operations that may occur in tasklist-based workflows. Let $(\mathsf{t}, r) \in TA$. Then delegation of task assignments of t may occur in (at most) three possible ways: to a user assigned to a less senior role (downward delegation), to the same role or to a more senior role (upward delegation).

6.1 Controlling delegation

For any access control mechanism, it is important to control user operations. Delegation operations such as downward delegation are well understood in the literature. Delegation operations that delegate a task to a user assigned to the same role to which the

[8] A *tasklist* is referred to as a *worklist* in the WfMC Workflow Reference Model.

[9] There may be several users who are authorized to perform a given task t. The particular choice of a user to whom t can be assigned to in a given tasklist, which is made by the workflow management system will depend on criteria such as work-load balancing, availability of the users, etc. Such considerations are beyond the scope of our discussion.

task is assigned are always possible and do not require any additional controls. However, it is the upward delegation that raises some interesting questions in tasklist-based workflow systems.

At first sight, it might appear that, while authorizing upward delegations, applying restrictions on the set of delegatee users is of little significance. On the contrary, we believe that it is more efficient to limit the set of delegatee users to ensure that tasks of lesser significance are not delegated to users who are authorized for more senior roles. In other words, with respect to the role hierarchy, the "gap" between the delegator and the delegatee must not be too large. As a trivial example, assume that (`raise_Cheque,Clerk`) $\in TA$. Now, should a request by a user assigned to a `Clerk` role to delegate a task of `raise_Cheque` to a user assigned to a `General_Manager` be permitted? In most cases, we do not wish to permit such delegations. However, we may certainly wish to allow an upward delegation of a task to a user assigned to an immediate senior role (or perhaps the delegator's line manager). Note that none of the existing works that study delegation in workflow systems provides support for appropriate controls while permitting upward delegation of tasks [25–28]. In this section, we will demonstrate how to control delegation operations in tasklist-based workflows.

Recall that, in Section 3.2, we have shown that the use of administrative scope for controlling delegations is more effective than using delegation relations. Hence, in this section, we discuss the application of the control model developed in Section 3.2 (that is based on the concept of administrative scope) for authorizing delegation of tasks in tasklist-based workflows.

Given a request by u to delegate task t (currently assigned to v in a tasklist) to a user w, the following questions should be considered by a reference monitor:

– Is u authorized to update a tasklist assignment containing (t, v) by delegating t to w?[10]
– Is w authorized to receive delegation of t?

As in Section 3.2, we limit the ability of a user to delegate tasks using the administrative scope of the roles that are assigned to the user. Specifically, a user u may delegate a task t if $(\mathtt{t}, r) \in TA$, $(u, r') \in UA$ and $r \in \sigma(r')$. That is, a user u can only delegate tasks that are assigned to roles within the administrative scope of roles assigned to u.

In order for the delegation of a task t to a user w to succeed, we require that w is "sufficiently" authorized, as usual. As in Section 3.2, the success of a downward delegation of t to w will depend on the role(s) to which `t` is assigned and the set of roles to which the delegatee user w is assigned. A downward delegation requires that the delegatee w is assigned to a role that is within the administrative scope of the role to which `t` is assigned. More formally, for a downward delegation of task t to w to succeed, we require that there exists a role $r' \in R$ such that $(w, r') \in UA$ and $r' \in \sigma(r)$. A delegation of task t to a user w assigned to a role $r \in R$ such that $(\mathtt{t}, r) \in TA$ and $(w, r) \in UA$ does not require any restrictions and is always possible.

6.2 Upward delegation

In RBAC96, a delegatee is not required to exercise a permission that has been delegated to him. However, in workflow execution models that are based on tasklists, there is a

[10] This is essentially delegation of a permission.

requirement for the delegatee to perform the delegated task. Hence, we require stronger conditions on the relationship between the delegator and the delegatee. Specifically, we need to ensure that the delegator cannot assign tasks to users for whom it would not be appropriate to perform the task.

A problem while developing a model for controlling upward delegation, with regard to the role hierarchy, is how do we decide the set of delegatee users to whom a task can be delegated? We may require that a task can only be delegated to users who are assigned to an immediate senior role of the delegation task. Such a control would be appropriate if we assume that each role is responsible for its immediate junior role. However, we believe that such an approach might be too restrictive. In most real world organizations, it might often be the case that a line manager is responsible for more than one role. Hence, we define two different upward delegation operations, namely a *weak* upward delegation and a *strong* upward delegation.

A *weak* upward delegation requires that the delegatee user is explicitly assigned to a role r' that belongs to the same administrative domain as the role r to which the delegation task is assigned and $r' < r''$. Let $D(r)$ be the smallest administrative domain to which a role $r \in R$ belongs, and let $D^*(r)$ be the parent domain of $D(r)$. Note that $D^*(r)$ is uniquely defined by Lemma 2. Let $R_u = \{r \in R : (u, r) \in UA\}$; that is, R_u denotes the set of roles to which u is explicitly assigned. We define $R_{\mathtt{t}}$ analogously. Then for user u to delegate an instance of $\mathtt{t}$ to delegatee w, we require that there exist $r_{\mathtt{t}} \in R_{\mathtt{t}}$, $r_u \in R_u$ and $r_w \in R_w$ such that

$$r_{\mathtt{t}} \in \sigma(r_u); \tag{1}$$

$$r_w \in D^*(r_{\mathtt{t}}). \tag{2}$$

The first of these conditions, is the usual one for the "strength" of the delegator. Condition (2), however, is stronger than the one we used in Section 3.2. It has the effect of ensuring that the delegatee has at least one administrative domain in common with the task. In other words, the delegator cannot assign tasks to users who have no association with the administrative domain(s) associated with the task.

Consider, for example, Figure 1 and assume that a user u, where $R(u) = \{\mathtt{PL1}\}$, initiates a request to delegate an instance of a task $\mathtt{t} \in \mathsf{T}$, where $R(\mathtt{t}) = \{\mathtt{ENG1}, \mathtt{ENG2}\}$. Note that $\mathtt{ENG1} \in D(\mathtt{ENG1}), D^*(\mathtt{ENG1}) = D(\mathtt{PL1}), \mathtt{ENG2} \in D(\mathtt{ENG2})$ and $D^*(\mathtt{ENG2}) = D(\mathtt{QE2})$. Then the above weak upward delegation of t to a user w by u is permitted if w is explicitly assigned to a role $r' \in R$ such that $r' \in D(\mathtt{PL1})$. In other words, we require that w is assigned to at least one of the roles $\{\mathtt{QE1,PE1,PL1}\}$. Now let us assume that the delegator user $R(u) = \{\mathtt{PL1}, \mathtt{PL2}\}$. Then, we require that w is explicitly assigned to a role $r' \in R$ such that $r' \in \{D(\mathtt{PL1}) \cup D(\mathtt{QE2})\}$. That is, we require that w is assigned to at least one of the roles $\{\mathtt{QE1,PE1,PL1,QE2}\}$.

A *strong* upward delegation requires that the delegatee user is assigned to a role r' that is an *immediate senior* role to a role $r \in R$ such that $(\mathtt{t}, r) \in TA$. In the above example, if the delegator user u is assigned to only PL1 role then a strong upward delegation of t to a user w is permitted if w is explicitly assigned to at least one of the roles $\{\mathtt{QE1,PE1}\}$. However, if u is assigned to both roles PL1 and PL2 then, for a strong upward delegation of t to a user w to succeed, we require that w is explicitly assigned to at least one of the roles $\{\mathtt{QE1,PE1,QE2}\}$.

7 Conclusions and Future Work

We have developed a comprehensive delegation model for role-based access control that provides support for both grant and transfer delegation policies. This is the first attempt in the literature that extensively studies transfer delegations for role-based access control.

We have discussed two mechanisms for controlling delegations: the relation-based approach and the administrative scope approach. In particular, we have shown that the concept of administrative scope can be used for authorizing delegations and is more effective than relations for delegation in dynamic environments.

We have also presented an enforcement mechanism that supports both grant and transfer delegation policies. Our enforcement model uses a delegation history relation DH and temporary delegation relations, *tempUA* and *tempPA*, to guarantee that delegations are enforced correctly. We also discussed various effects on the above relations following a successful delegation operation and corresponding revocations. Furthermore, we have shown that the simple enforcement mechanism for revocation that has been described in this paper can also support more complex revocation policies such as those described in [19].

We have also shown the application of our delegation model for authorizing delegation operations in the context of tasklist-based workflow systems. Most importantly, we have shown the use of administrative scope for controlling downward task delegations and administrative domains for controlling upward task delegations. This is the first work in the literature that controls upward delegation of tasks in tasklist-based workflows.

An immediate priority in future work is to define necessary commands that provide an option to set the delegatable flag in the delegation mask and to set the values for more complex revocation techniques such as those described in [19]. We would also be very much interested in studying delegation in the context of constrained workflow systems. A long term goal is to develop abstract Java classes that implement our delegation model.

References

1. R. Sandhu, E.J. Coyne, H. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
2. A. Schaad. *A Framework for Organisational Control Principles.* PhD thesis, The University of York, York, England, 2003.
3. E. Barka and R. Sandhu. Framework for role-based delegation models. In *Proceedings of Twenty Third National Information Systems Security Conference (NISSC'00)*, pages 101–114, 2000.
4. T. Aura. Distributed access-rights management with delegation certificates. In *Secure Internet Programming – Security Issues for Distributed and Mobile Objects*, volume 1603 of *LNCS*, pages 211–235. Springer, 1999.
5. V.D. Gligor, S.I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 172–183, 1998.

6. R. Simon and M. Zurko. Separation of duty in role-based environments. In *Proceedings of Tenth IEEE Computer Security Foundations Workshop*, pages 183–194, 1997.
7. E. Barka and R. Sandhu. A role-based delegation model and some extensions. In *Proceedings of Sixteenth Annual Computer Security Applications Conference (ACSAC'00)*, pages 168–177, 2000.
8. S. Na and S. Cheon. Role delegation in role-based access control. In *Proceedings of Fifth ACM Workshop on Role-Based Access Control (RBAC'00)*, pages 39–44, 2000.
9. J. Park, Y. Lee, H. Lee, and B. Noh. A role-based delegation model using role hierarchy supporting restricted permission inheritance. In *Proceedings of the 2003 International Conference on Security and Management (SAM'03)*, pages 294–302, 2003.
10. R. Tamassia, D. Yao, and W. Winsborough. Role-based casdaded delegation. In *Proceedings of Ninth ACM Symposium on Access Control Models and Technologies (SACMAT'04)*, pages 146–155, 2004.
11. J. Wainer and A. Kumar. A fine-grained, controllable, user-to-user delegation method in RBAC. In *Proceedings of Tenth ACM Symposium on Access Control Models and Technologies (SACMAT'05)*, pages 59–66, 2005.
12. L. Zhang, G.-J. Ahn, and B.-T. Chu. A rule-based framework for role-based delegation and revocation. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):404–441, 2003.
13. X. Zhang, S. Oh, and R. Sandhu. PBDM: A flexible delegation model in RBAC. In *Proceedings of Eighth ACM Symposium on Access Control Models and Technologies (SACMAT'03)*, pages 149–157, 2003.
14. J. Crampton and G. Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information and System Security (TISSEC)*, 6(2):201–231, 2003.
15. D. Ferraiolo, D.R. Kuhn, and S. Chandramouli. *Role-Based Access Control*. Artech House, Boston, Massachussetts, 2003.
16. R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 1(2):105–135, 1999.
17. J. Crampton. Understanding and developing role-based administrative models. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 158–167, 2005.
18. E. Barka. *Framework for Role-Based Delegation Models*. PhD thesis, George Mason University, Virginia, USA, 2002.
19. Å. Hagström, S. Jajodia, and F. Parisi-Presicce. Revocations – A classification. In *Proceedings of the Fourteenth IEEE Workshop on Computer Security Foundations (CSFW'01)*, pages 44–58, 2001.
20. D. Hollingsworth. Workflow management coalition: The workflow reference model, 1995. Document Number TC00-1003, Document Status- Issue 1.1, Available at `http://www.wfmc.org/standards/docs/tc003v11.pdf`.
21. IBM WebSphere MQ Workflow version 3.6. Available at `http://www-306.ibm.com/software/integration/wmqwf/`.
22. BEA AquaLogic BPM (ALBPM) version 5.7. Available at `http://edocs.bea.com/albsi/docs57/index.html`.

23. E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):65–104, 1999.
24. S. Kandala and R. Sandhu. Secure role-based workflow models. *Database Security XV: Status and Prospects*, pages 45–58, 2002.
25. K. Venter and M. Olivier. The delegation authorization model: A model for the dynamic delegation of authorization rights in a secure workflow management system. In *Proceedings of Information Security South Africa (ISSA'02)*, 2002. Published electronically. Available at `http://icsa.cs.up.ac.za/issa/2002/proceedings/A021.pdf`.
26. V. Atluri, E. Bertino, E. Ferrari, and P. Mazzoleni. Supporting delegation in secure workflow management systems. In *Proceedings of Seventeenth Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 190–202, 2003.
27. V. Atluri and J. Wainer. Supporting conditional delegation in secure workflow management systems. In *Proceedings of Tenth ACM Symposium on Access Control Models and Technologies (SACMAT'05)*, pages 49–58, 2005.
28. J. Wainer, A. Kumar, and P. Barthelmess. DW-RBAC: A formal security model of delegation and revocation in workflow systems. *Information Systems*, 32(3): 365–384, 2007.