Specifying and Enforcing Constraints in Role-Based Access Control

Jason Crampton

Information Security Group
Royal Holloway, University of London
Egham, TW20 0EX, United Kingdom

jason.crampton@rhul.ac.uk

ABSTRACT

Constraints in access control in general and separation of duty constraints in particular are an important area of research. There are two important issues relating to constraints: their specification and their enforcement. We believe that existing separation of duty specification schemes are rather complicated and that the few enforcement models that exist are unlikely to scale well.

We examine the assumptions behind existing approaches to separation of duty and present a combined specification and implementation model for a class of constraints that includes separation of duty constraints. The specification model is set-based and has a simpler syntax than existing approaches. We discuss the enforcement of constraints and the relationship between static, dynamic and historical separation of duty constraints. Finally, we propose a model for a scalable role-based reference monitor, based on dynamic access control structures, that can be used to enforce constraints in an efficient manner.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; I.6.0 [**Computing Methodologies**]: Simulation and Modeling

General Terms

Security, Theory

Keywords

role-based access control, separation of duty constraint, authorization constraint, enforcement context

SACMAT'03, June 2–3, 2003, Como, Italy.

1. INTRODUCTION

Separation of duty is an important control principle in management whereby sensitive combinations of duties are partitioned between different individuals in order to prevent the violation of business rules. A very simple example of this is that cheques might require two different signatures. In a military context, we might require that two different individuals must independently arm and launch a nuclear missile.

The research community has taken an active interest in incorporating separation of duty controls into computer systems since the late 1980s. One of the rules of the Clark-Wilson model [6] requires that separation of duty requirements must be met. Since then, several papers have studied separation of duty. One of the best known requirements for separation of duty is embodied in the Chinese Wall model [5], in which access to documents that could result in a commercial conflict of interest is strictly controlled. It was suggested that the Chinese Wall model could be implemented by maintaining a history matrix indexed by subjects and objects, whose (binary) entries indicated which documents had been accessed by which users.

Role-based access control models have attracted considerable research interest in recent years due to their innate ability to model organizational structure and their potential to reduce administrative overheads. An important feature of role-based models has been the specification of separation of duty constraints [10, 12, 13, 15, 16].

The seminal paper of Simon and Zurko on separation of duty in role-based systems [16] proposed a rule-based specification scheme for separation of duty constraints and a history-based implementation to enforce those constraints. Several authors have since identified increasingly complex separation of duty requirements and specification schemes to express them [2, 10, 11, 12]. None of these papers have suggested a model for implementing such constraints.

Separation of duty requirements are an important issue in workflow management systems. Bertino *et al.* suggest that such requirements can be enforced using logic programming techniques to compute all valid execution paths for a workflow (in the presence of constraints) and permitting an access request to proceed only if it belongs to a valid execution path [4]. It is our belief that this approach will not scale well to large-scale applications.

In this paper we propose a simple specification scheme

for separation of duty constraints. In fact, we can define constraints that are not separation of duty constraints in the traditional sense. Therefore, we will use the term *authorization constraint* [4]. Unlike most existing specification schemes, we do not explicitly specify the conditions that must be preserved if the constraint is to be satisfied. This is for two reasons: firstly, we believe that this approach places an unnecessary burden on the syntax of the specification scheme; and, secondly, that the constraints should be enforced by the reference monitor. Therefore, we also propose a simple implementation model that can be used to enforce a restricted set of the authorization constraints supported by our specification scheme. This model is based on the idea of dynamic access control structures that are updated when constraint-relevant events occur.

The major contributions of this paper are to provide a simple specification scheme for role-based constraints and a novel implementation model that should prove more scalable than existing approaches. While the specification scheme is slightly less expressive than the most recent alternatives [2, 12], we believe that our approach is far simpler to understand and has a much less cumbersome syntax. Moreover, our implementation scheme is simpler than the one proposed by Simon and Zurko [16], while the specification scheme supports a greater range of constraints with a simpler syntax than their model.

In the next section, we make some preliminary observations about separation of duty constraints in order to motivate our approach. In Section 3, we will define our specification scheme and provide a comprehensive list of examples. In Section 4, we introduce our enforcement model and some illustrative examples. In Section 5, we discuss related work, compare it to our approach, and then discuss some shortcomings of our approach. Finally, we summarize the contributions of the paper and discuss future work.

2. BACKGROUND

We will introduce a role-based access control model to provide a context for the specification scheme and enforcement model. We will also discuss constraints in role-based models and workflow systems.

We then discuss some important issues related to constraint specification in a role-based model. This discussion will provide the motivation for our specification scheme and for certain aspects of the enforcement model.

2.1 RBAC96

We develop the material in this paper in the context of RBAC96, the most widely known role-based access control model [15]. (In fact, RBAC96 consists of four different models, RBAC_0, RBAC_1, RBAC_2 and RBAC_3, the last two of which incorporate separation of duty constraints.) RBAC96 identifies the following sets: U, a set of users; R, a partially ordered set of roles, which is usually interpreted as a role hierarchy $RH \subseteq R \times R$; P, a set of permissions; $UA \subseteq U \times R$, a user-role assignment relation; and $PA \subseteq P \times R$, a permission-role assignment relation.

2.2 Permissions

Although some researchers regard permissions as "uninterpreted symbols" [9], we believe that is helpful to give them some semantics in order to make some of the constraints we discuss more concrete. We take the conventional view that a permission is an ability to do something to an object [16]. Hence we define an *abstract permission* to be a pair (t, a) where t is a *type* and a is an *action*. A permission is a pair (o, a), where o is an instance of the type t.[1] Permissions and abstract permissions are assigned to roles. The assumption is that a new object o (of type t) is created in the system by a role invoking the abstract permission (t, new), where *new* is the create action (constructor method) for the type t. At this point any role with permissions defined for that type obtains additional permissions for the particular instantiation of that type, which could include (o, set) ("write") and (o, get) ("read"), for example.

2.3 RBAC constraints

The existence of a role hierarchy facilitates the specification of separation of duty constraints because of its ability to model organizational structures. For example, a purchase order clerk role and a finance clerk role will typically be incomparable roles in the role hierarchy, and it may well be an organizational requirement that no user be assigned to both roles.

In a role-based access control environment, there are six sets of entities that can be part of a separation of duty constraint: users (U), roles (R), permissions (P), objects (O), types (T) and actions (A).[2] For example, given a *cheque* type for which the actions *raise* and *issue* are defined, we may wish to constrain the application so that the same user (acting in the role of payment clerk, say) cannot invoke both actions on the same cheque object. This is an example of a separation of duty constraint defined for users and permissions.[3]

In a workflow environment, it may be that the *approve* action for the *cheque* type must be invoked twice by different users and by a role that is more senior than the role that invoked the *raise* action [4]. Alternatively, we may have an *approve* action and insist that the permissions $(cheque, approve)$ and $(cheque, raise)$ are not assigned to the same role.

The literature has long recognized that there are three different "flavours" of separation of duty constraint. *Static separation of duty* typically constrains the assignment of users and permissions to roles. *Dynamic separation of duty* typically constrains the activation of roles and invocation of permissions in the run-time environment. *Historical separation of duty* typically constrains the invocation of permissions over the course of time; for example, we may require that no user can raise and issue the same cheque.

In many cases, a constraint will apply to the same object, as in the cheque example above. Several papers have distinguished between order-dependent and order-independent

[1] This interpretation of permissions has a natural implementation in an object-oriented application where a type corresponds to a class and an object is an instantiation of a class. (It would also be possible to view a permission as some form of query on a database table or view.)

[2] A constraint defined on a type is assumed to apply to all instances of the type.

[3] In a more sophisticated model, we might assume that an abstract permission is a triple $(type, domain, action)$, where the *domain* identifies a subset of objects that are instances of the type. We can then define a separation of duty constraint on the permissions $(account, company_a, read)$ and $(account, company_b, read)$ in order to specify a Chinese Wall style policy.

constraints [12, 16]. In the former case, the constraint would require that a cheque is raised before it is issued. In the latter case, no such requirement would be made.

While the distinction between these cases seems superficially attractive, we believe that the life cycle of an object will be determined by the business rules and will be encoded in the application. In our example, it is unlikely that a financial application would permit a non-existent cheque (one that has not been raised) to be issued. We believe, therefore, that it is unnecessary to be able to express order-dependent requirements within a constraint specification scheme. We will discuss the implications of this decision when we review related work.

2.4 Observations

We now make some observations about separation of duty that summarize the contents of this section and indicate how they will inform our approach.

- Separation of duty is not a complicated concept!

  Separation of duty requirements articulate circumstances that would lead to the violation of business rules. A violation could be regarded as the occurrence of a set of events that contravene the business rules. In other words, to specify a separation of duty constraint we merely need to enumerate the "undesirable" set(s) of events, and to enforce a constraint we must ensure that all the events specified in that set cannot occur.

- A specification scheme must be straightforward to use: it is the users of an application who will be required to specify separation of duty constraints.

  Research papers on separation of duty in computer systems regularly describe constraints that are defined in terms of users and in terms of roles. It is not immediately obvious when these constraints are to be specified. Consider, for example, an integrated purchasing and bought ledger application that raises purchase orders for goods and services and also pays the resulting invoices. There are certain separation of duty constraints that could be hard-wired into the application – no user can be assigned to both the purchase order clerk and finance clerk roles, for example. However, there are certain separation of duty constraints that can only be specified by the administrators of a particular installation of that application.

- A specification scheme must include historical constraints.

  Static separation of duty is not a practical or realistic variation of separation of duty because it does not capture most real-world organisational control principles [16].

- We assume that historical constraints are order-independent.

  That is, we assume the business logic encoded in the application imposes an ordering on execution.

3. THE SPECIFICATION OF CONSTRAINTS

Informally, a constraint defines a family of "bad" sets. Enforcement of a constraint requires that for all bad sets, it is never the case that all elements in a bad set can occur. For example, a static separation of duty constraint might require that no user can be assigned to both roles r_1 and r_2. In this case, the "bad" sets are $\{(u_1, r_1), (u_1, r_2)\}, \ldots, \{(u_n, r_1), (u_n, r_2)\}$, where $U = \{u_1, \ldots, u_n\}$. (The constraint will be enforced by considering the UA relation.)

3.1 Formal description

We now formally define our specification scheme. A *constraint* is a triple (s, c, x), where s is the (*constraint*) *scope*, c is the *constraint set* and x is the (*temporal*) *context* and takes one of the following values: *static*, *dynamic* and *historical*, which are denoted s, d and h, respectively. The scope and the constraint set are subsets of one of the following sets: U, R, P, T and O. The static separation duty example in the introduction to this section is written $(U, \{r_1, r_2\}, s)$.

A constraint defines a family of sets: in particular, the constraint $c = (A, B, x)$ defines the family of sets

$$\mathcal{Q}_c = \bigcup_{a \in A} (\{a\} \times B).$$

Each $Q \in \mathcal{Q}_c$ is called a *constrained* (*authorization*) *set*. Each $q \in Q$, where Q is a constrained set, is called a *constrained* (*authorization*) *request*. For example, if $(U, \{p_1, p_2\}, h)$ is a constraint, then for each $u \in U$, $\{(u, p_1), (u, p_2)\}$ is a constrained set and (u, p_i) is a constrained request.

A constrained authorization set is a subset of $X \times Y$, where $X, Y \in \{U, R, P, T, O\}$. $X \times Y$ is called the (*constraint*) *enforcement context*. For example, the enforcement context of $(U, \{r_1, r_2\}, s)$ is UA. We will discuss this in more detail in Section 4.1.

3.2 Examples

We now enumerate some of the varieties of separation of duty that have been identified in the literature and that can be expressed in our scheme. For comparative purposes, we indicate the correspondence between our examples and those in the paper by Jaeger and Tidswell [12], which provides probably the most comprehensive set of examples in the literature.[4]

We do not enumerate all the possibilities available within our scheme. In particular, we do not generally discuss the three constraints that are possible by considering the different temporal contexts. We implicitly assume the reader will appreciate the different semantics of each of these constraints. We will focus on constraints that have received attention in the literature or ones that we believe are novel and have some useful application.

[4]The RCL 2000 constraint specification language [2] and the specification scheme proposed by Jaeger and Tidswell admit the articulation of certain constraints that are not possible in our scheme. These constraints are based on the aggregation of users and permissions with quantification over sets and members of sets. However, we believe that our proposal is simpler to understand; we will also provide a model for implementing our constraints, which, with the exception of the Adage implementation [16], has not been attempted in previous work. It should also be noted that Jaeger and Tidswell's scheme has been designed to be applicable in a general access control model, not just in role-based access control models. We believe that our scheme can be extended easily to other access control models.

3.2.1 User-based separation of duty

In this case, the constraint set is a subset of U.

- The constraint $(R, \{u_1, u_2\}, s)$ is a separation of duty constraint that requires that no role is assigned to both r_1 and r_2 [12, Example 1].
- The constraint $(\{r_1, \ldots, r_n\}, \{u_1, u_2\}, s)$ is a constraint that requires that no role in a specified set (the scope $\{r_1, \ldots, r_n\}$) is assigned to both u_1 and u_2 [12, Example 8].

3.2.2 Role-based separation of duty

In this case, the constraint set is a subset of R, where RH is the partially ordered set of roles $\langle R, \leqslant \rangle$. Constraints such as these were among the first identified in the literature.

- The constraint $(U, \{r_1, r_2\}, s)$ is a simple static separation of duty constraint that requires that no user is assigned to both r_1 and r_2 [12, Example 3].
  - $(U, \{r_1, r_2\}, d)$ specifies the corresponding dynamic separation of duty constraint.
  - The constraint $(U, \{r_1, \ldots, r_n\}, s)$ requires that no user is assigned to all the roles in the constraint set.
- The constraint $(\{u_1, \ldots, u_n\}, \{r_1, r_2\}, s)$ is a similar constraint in which no member of a specified set of users can be assigned to both r_1 and r_2 [12, Example 8].
  - Another interesting possibility is to use a role as a placeholder for a set of users. That is we define a role r that has no permissions assigned to it and use it to define a group of users. We denote the set of users assigned to a role r by $U(r)$. That is, $U(r) = \{u \in U : (u, r) \in UA\}$. We can then define a constraint $(U(r), \{r_1, r_2\}, s)$, which states that no user in the group associated with r can be assigned to both r_1 and r_2.
- The constraint $(\{u_1, \ldots, u_n\}, \{r\}, s)$ is a constraint that requires that no user in a specified set can be assigned to the role r [12, Example 9].

  Constraints such as these are rarely supported by existing specification schemes. However, there are a number of practical examples where such constraints are useful.

  For example, in the role graph model it is generally assumed that no user should be assigned to `MaxRole` [13]; $(U, \{\texttt{MaxRole}\}, s)$ expresses this constraint succinctly. In other words, we can express role exclusion constraints: such constraints can also be used to impose a "ceiling" on the roles to which a user can be assigned. For example, the constraint $(u, \{r\}, s)$ prevents user u from being assigned to the role r (or any role more senior than r). (Similar constraints on permissions can be used to impose a "floor" on the set of roles to which a permission can be assigned.)
- The requirement that no user is assigned to more than one role in a specified set $\{r_1, \ldots, r_n\}$, say, can be expressed as $\binom{n}{2}$ constraints of the form $(U, \{r_i, r_j\}, s)$, $1 \leqslant i < j \leqslant n$ [12, Example 12].
- The constraint $(\{p\}, \{r_1, r_2\}, s)$ is a static constraint that requires that p cannot be assigned to both r_1 and r_2.

  We can envisage domain-based requirements where permissions for certain mutually exclusive activities should be separated in this way. The Chinese Wall model [5] is a potential application of such a constraint.

3.2.3 Permission-based separation of duty

In this case, the constraint set is a subset of P, the set of permissions.

- The constraint $(U, \{p_1, p_2\}, s)$ is a static separation of duty constraint that requires that no user can invoke both permission p_1 and p_2.

  An example of why such a constraint might be useful arises if we do not want any user to be able to raise both a purchase order and a cheque.
  - The constraint $(U, \{p_1, p_2\}, h)$ is a historical version of the previous separation of duty constraint [12, Example 4].
    Such a constraint could be used to specify that no user can both raise and issue a cheque.
- The constraint $(U, \{p_1, \ldots, p_n\}, c)$ is a constraint that requires that no user can invoke all the permissions to complete some task (where the execution of a task is assumed to be equivalent to the invocation of a sequence of permissions). This constraint can be static [12, Example 10], dynamic [12, Example 11] or historical (as discussed by Bertino *et al.* in their analysis of workflow systems [4]).
- The constraint $(R, \{p_1, p_2\}, h)$ is a historical constraint that requires that no role can invoke both p_1 and p_2 [12, Example 2].

  We can also specify constraints of the form $(R, \{p_1, p_2, p_3\}, s)$, which states that a role can be assigned to any two of the permissions in the constraint set (but not all three). Bertino *et al.* [4] have identified certain workflow applications that require such constraints.

3.2.4 Object-based separation of duty

In this case, the constraint set is a subset of O, the set of objects, or of T, the set of types.

- The constraint $(U, \{o_1, o_2\}, h)$ is a historical constraint that no user can ever access both object o_1 and o_2 [12, Example 6].

  This constraint could be used to implement the Chinese Wall model.
- The constraint $(R, \{o_1, o_2\}, h)$ is a similar constraint where the scope is the set of roles rather than the set of users [12, Example 5].

  This is a weaker constraint than the one above because we could have o_1 and o_2 associated with roles r_1 and r_2, respectively. Then any user assigned to both these roles can access both objects.
- The constraint $(P, \{o_1, o_2\}, h)$ is an even weaker constraint where the scope of the constraint is the set of permissions, which does not prevent either a role or a user having access to both o_1 and o_2.

4. ENFORCING CONSTRAINTS IN A ROLE-BASED SYSTEM

A *constrained role-based system* (hereafter simply referred to as a system) is a role-based system for which a set of constraints C is defined. We define the *configuration* of a system to be the tuple (UA, PA, RH, C). We define the *state* of a system to be the tuple $(UA, PA, RH, C, UA_I, P_I)$, where $UA_I \subseteq UA$ denotes the set of user sessions (each of these sessions is a set of roles activated by a user) and $P_I \subseteq U \times P$ denotes the set of permissions that are currently invoked. (For each $(u, p) \in P_I$, there exists $(u, r) \in UA_I$ such that $(p, r) \in PA$.)[5]

A *request* is an interaction by a user with a system that potentially results in a change to the configuration or state of the system. In a role-based context, such requests include: a request to invoke a permission; a request to activate a role or to establish a session; attempts to assign a user or a permission to a role; and changes to the role hierarchy. An authorization constraint c is *enforced* if for all $Q \in \mathcal{Q}_c$, it is not possible for all $q \in Q$ to be granted. We will assume that constraints are enforced by a *constraint monitor*, which can be viewed as an additional component of the reference monitor. We will describe possible implementations of the constraint monitor in more detail in the Sections 4.2 and 4.3.

4.1 The enforcement context of a constraint

Constraints can be enforced either by the configuration or the state or the state history of a system. These are usually referred to as static, dynamic and historical constraints, respectively.

For example, $(U, \{r_1, r_2\}, x)$ is a typical constraint which requires that no user be assigned both r_1 and r_2. If this is a static constraint, then it is satisfied if for all users u, at most one of (u, r_1) and (u, r_2) belongs to UA (that is, $\{(u, r_1), (u, r_2)\} \not\subseteq UA$). If, however, it is a dynamic constraint, then it is satisfied if for all users u, $\{(u, r_1), (u, r_2)\} \not\subseteq UA_I$. (That is, no user has activated both r_1 and r_2. Note that it is possible in this case for a user to be assigned to both roles, but the user will not be permitted to activate both in the same session.)

However, it is difficult to interpret this as a historical constraint. Should it mean that once u has been assigned to r_1 (say), then u can never be assigned to r_2. Alternatively, should it mean that once u has activated r_1, then u can never activate r_2. These interpretations of enforcement context could be classified as *static historical* and *dynamic historical*, respectively. To the author's knowledge, no such distinction exists in the literature. A detailed discussion of these constraints and whether we need additional historical interpretations of them is beyond the scope of this paper. We believe that the dynamic historical interpretation is likely to be more meaningful in practice, and is the interpretation we shall adopt in the remainder of this paper.

It is easily seen that the enforcement of a static authorization constraint implies that the corresponding dynamic constraint is enforced. For example, if the static constraint $(U, \{r_1, r_2\}, s)$ is enforced, then no user can possibly activate both roles (since a user must be assigned to a role before it can be activated).

However, the dependencies between historical and static enforcement are less clear. In fact, if a static constraint is enforced, then the following counterexample shows we cannot infer that the corresponding historical constraint is enforced. Suppose that u is assigned to r_1, then has the assignment revoked and is subsequently assigned to r_2. (Note that this necessarily violates the static historical constraint.) The static constraint has not been violated, but it is possible that u has activated both r_1 and r_2 during the periods that u was assigned to the respective roles, violating the dynamic historical constraint.

Similarly, we cannot infer that the enforcement of a dynamic historical constraint implies that the corresponding static constraint has been enforced. However, we can assert that the enforcement of a static historical constraint implies that the corresponding static constraint has been enforced. Figure 1 shows the relationships that exist between the different enforcement contexts. (An arrow between nodes indicates that the enforcement in the context at the head of the arrow is implied by enforcement in the context at the tail of the arrow.)

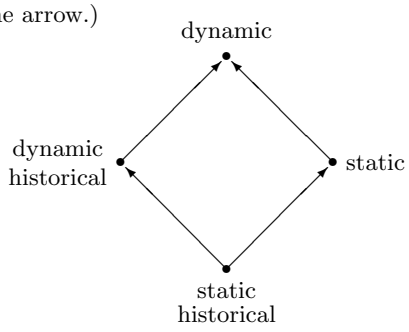

Figure 1: The relationship between enforcement contexts

4.2 Blacklists: Enforcing historical constraints

In order to enforce a historical authorization constraint it may be necessary to deny certain requests. For example, if $(U, \{p_1, p_2\}, h)$ is a constraint and u has invoked permission p_1, then a request by u to invoke permission p_2 must be denied in order to enforce the constraint.

One possible approach to enforcement is to employ a historical record [5, 14, 16] of all previous invocations of permissions and to consider this record whenever a request to invoke a permission is made. The Chinese Wall model defines the history matrix for precisely this purpose. It is our belief that approaches of this nature will not scale well to large-scale applications.

We employ a different approach to the enforcement of historical constraints by dynamically changing the requests that can succeed. Hence, in order to enforce a constraint, we create a *blacklist* – a dynamic access control structure that contains constrained requests. When a constrained request occurs the relevant blacklist is consulted. If the request belongs to the blacklist, the request is denied; otherwise, the request is referred to the role-based reference monitor.

Let $(U, \{p_1, p_2\}, h)$ be a constraint. If u successfully invokes permission p_1, then a blacklist is created containing (u, p_2). If u now tries to invoke permission p_2, then the

[5]For every $s \in S$, the set of roles in s is a subset of the roles implicitly assigned to the user who activated s [15]. The set P_I is similar to the set of currently granted subject-object-access triples that forms part of the definition of state in the Bell-LaPadula model [3]. (This set is used to determine whether a request will violate the *-property.)

request fails. However, if u has not already invoked p_1, the request is referred to the reference monitor to establish whether u has activated a role r such that $(p_2, r) \in PA$ and the request can be granted.

We now consider the implementation of blacklists in more detail. The literature suggests that the most important historical authorization constraints are concerned with users and permissions. Therefore, we will focus on how such constraints can be enforced using blacklists and then comment more briefly on how other historical constraints can be enforced.

We restrict our attention to constraint sets that contain no more than two elements. Therefore, we can only specify and enforce standard separation of duty constraints and exclusion constraints. We will briefly discuss how the enforcement model might be extended to accommodate more complex constraint sets in Section 5.

4.2.1 User-permission constraints

The constraint $(U, \{p_1, p_2\}, h)$, defines the constrained sets $\{(u, p_1), (u, p_2)\}$, where $u \in U$. If user u invokes permission p_1 successfully, any subsequent request to invoke permission p_2 must be denied if the constraint is to be enforced. Therefore, we could include (u, p_2) in a blacklist associated with the constraint monitor. However, we believe a more efficient approach is to create a list of prohibited permissions associated with the user u. In other words, we maintain a blacklist for each user consisting of a set of permissions. However, note that a set of permissions is nothing more than a role, so we can view this blacklist as an "anti-role". Therefore, each user u is assigned to a role ρ_u containing permissions that are currently prohibited for that user.

The work of Sandhu on *transaction control expressions* [14] assumed it was possible to define a life cycle for *transient* objects. (In the case of a *cheque* object, for example, this life cycle begins when the cheque is raised and ends when the cheque is issued.) That is, certain permissions are necessarily only invoked a certain number of times. (In the case of business objects, there will often only be a single invocation.) Hence, once a permission ceases to become meaningful, it can be deleted from any blacklist it belongs to. (This is analogous to the process by which an object becomes *sanitized* in the Chinese Wall model [5]. Hereafter we will apply this term to permissions.) The process of sanitizing permissions and deleting them from a blacklist is not strictly necessary, but it does mean that redundancy in a blacklist is eliminated and that by minimizing the length of the blacklist it is easier to check for entries in the blacklist.

For example, let $(U, \{a_1, a_2\}, h)$ be an authorization constraint, where a_1 and a_2 are abstract permissions. We will assume that an object of the type associated with these abstract permissions is sanitized once both these permissions have been invoked. We also assume that p_1 and p_2 are the corresponding permissions associated with an object of this type. Consider the following sequence of requests: $Invoke(u, p_1)$, $Invoke(u, p_2)$ and $Invoke(v, p_2)$. Table 1 shows the effect of this sequence of events on ρ_u (which is initially empty) and whether each of the requests succeeds. Note that the assumptions we have made suggest that a role-based access control system should also periodically purge the PA relation of sanitized permissions (since their invocation no longer has any meaning).

Request	Decision	ρ_u
$Invoke(u, p_1)$	✓	$\{p_2\}$
$Invoke(u, p_2)$	✗	$\{p_2\}$
$Invoke(v, p_2)$	✓	$\emptyset$

Table 1: Enforcing the constraint $(U, \{p_1, p_2\}, h)$

4.2.2 Role-permission constraints

Bertino *et al.* identified certain situations in workflow management systems where constraints of the form $(R, \{p_1, p_2\}, h)$ are required [4]. The interpretation of such a constraint is that once a user has activated a role r and invoked permission p_1, then permission p_2 cannot be invoked by any user activating role r. One example might be that the purchase order clerk role can raise a purchase order but not sign for receipt of the goods delivered in respect of that order. This constraint could be used to prevent collusion amongst purchase order clerks, for example.

In order to implement a blacklist for this constraint, we would assign negative permissions to the role r. Any attempt to invoke a permission p using role r would be denied if the corresponding negative permission (which we denote $\neg p$) were assigned to r (irrespective of whether p was assigned to r). In other words, negative permissions always take precedence over normal permissions.

The standard interpretation of the assignment $(\neg p, r)$ in role-based access control is that $\neg p$ is implicitly assigned to every $r' \in R$ such that $r' < r$ [15]. With this interpretation, we can create an anti-role $\neg r$ for the role r, where $r < \neg r$, and assign negative permissions to $\neg r$, instead of assigning negative permissions to r.

Bertino *et al.* also consider constraints based on the structure of the role hierarchy. Again using the assumption that negative permissions are inherited downwards, we believe our model can be used to implement some of the constraints that have been identified. This is a matter for further research.

4.3 The constraint monitor

We assume the existence of a *constraint monitor* that is responsible for enforcing authorization constraints. Each constrained request could potentially cause a violation of an authorization constraint. Hence each instance of a constrained request is passed to the constraint monitor. The constraint monitor checks whether granting the request would violate an authorization constraint and takes appropriate action.

Note that it is possible to enforce a static constraint by considering the configuration of the system. In other words, we only require blacklists for historical authorization constraints. Similarly, it is possible to enforce a dynamic constraint by considering the state of the system. For example, suppose a user attempts to start a session by activating two roles r and r', which form a dynamic separation of duty constraint. Then the security monitor would prevent the session from starting.

However, it may be more efficient and also facilitate an object-oriented implementation if we handle all constraints using blacklists. For example, if $(U, \{r_1, r_2\}, s)$ is a constraint, u has already been assigned to r_1 and a request is made to assign u to r_2, it is likely to be more efficient to check whether (u, r_2) belongs to a blacklist than to check

whether $(u, r_1) \in UA$.

Hence we envisage that there could be several different classes of constraint monitors derived from some abstract constraint monitor class. These could include a role hierarchy monitor class, a user-role assignment monitor class, a permission-role assignment monitor class, a session monitor class and monitor classes for specific types (such as the *cheque* type). Each monitor will maintain a list of authorization constraints that are relevant to that monitor.

5. EVALUATION AND RELATED WORK

Specification. The specification scheme outlined in this paper has its basis in our set-based approach to conflict of interest policies [7]. It is similar to the model developed by Jaeger and Tidswell [12], which uses set predicates to define separation of duty constraints. The scheme we propose is considerably simpler syntactically than their scheme because we make no attempt to define the conditions that must be met for the constraint to be satisfied. Indeed, with the exception of the work of Simon and Zurko [16], most previous specification schemes for separation of duty have followed this approach and used some fragment of first-order logic as the specification language. We believe that it is well understood when a separation of duty constraint is violated and that including the conditions that would cause a violation simply increases the number of predicates and functions required to specify the constraints.

However, there are certain separation of duty constraints that can be specified in RCL 2000 [2] and in Jaeger and Tidswell's model that we can not specify using our approach. Broadly speaking these are constraints where it is not sufficient to iterate over the constraint set for each element of the scope. Recall that a constraint $c = (A, B, x)$ defines the family of sets

$$\bigcup_{a \in A} \{a\} \times B.$$

Hence it is not possible to express the following separation of duty requirement: given two users u and v and two roles r and s, we do not want either u or v to be assigned to both roles or u to be assigned one role and v assigned to the other. Such a constraint is potentially useful in preventing collusion between groups of users. A little thought shows that such a constraint defines the following constrained sets: $\{(u, r), (u, s)\}$, $\{(v, r), (v, s)\}$, $\{(u, r), (v, s)\}$ and $\{(u, s), (v, r)\}$. In other words, we could introduce a fourth parameter in our definition of constraint which determines the "direct product" semantics of the constraint: that is, whether we should iterate over both the scope and the constraint set or simply over the constraint set (as we currently do). (The Jaeger and Tidswell scheme includes eight different ways of iterating through the various components in a constraint.)

We also made an assumption that we did not need to consider order-dependent historical constraints. However, if it proved that such an assumption were not valid, it should be possible to interpret the constraint set as a constraint list. When an object is instantiated for which such a constraint is defined, we enter all elements of the list except the head of the list (the next event that should be executed) into the appropriate blacklist(s). Once that event occurs, the next element in the list is removed from the blacklist(s). Of course, this scheme is only applicable to linear workflows: if the workflow branches, we need to model a partial order. It is not immediately obvious how such workflow constraints could be specified within our scheme.

Enforcement. Dynamically modifying access control structures in order to reflect previous access requests or execution paths has received attention in several recent research papers. Edjlali *et al.* proposed a dynamic approach to controlling the access rights of mobile code in order to enforce requirements of the following form: if an application has accessed a file on the local host system, then the application can not open a socket [8]. More recently Abadi and Fournet have proposed an alternative to the "stack walk" semantics for virtual machines using the intersection of access rights that have been available to each process [1]. We have used these ideas as a starting point to develop the idea of a blacklist, which dynamically limits the permissions available to users (and possibly roles). (The concept of a blacklist also employs the concept of negative permissions, which have received little attention since their introduction to the role-based access control literature [15].)

The enforcement model we define in this paper can only enforce constraints in which the constraint set has no more than two elements. We do this because it is not possible to implement blacklists with a simple semantics otherwise. To see this, consider the constraint $(U, \{p_1, p_2, p_3\}, h)$ and assume that none of the three permissions have been invoked by user u. Once u invokes the permission p_1, say, which of the remaining two permissions should be entered in the blacklist? Clearly, we could enter both p_2 and p_3 into the blacklist ρ_u, but this would be too restrictive. Specifically, the semantics of constraint enforcement become that no more than one constrained request in a constrained set is permitted to succeed.

The alternative is to keep a historical record of all access requests and to enter either p_2 or p_3 at some point in the future. We do not want to keep a historical record because we believe that such an approach will have unacceptable performance overheads. Therefore, we have chosen to impose this upper bound on the cardinality of the constraint set in historical authorization contraints. It should be noted that most existing approaches to separation of duty only consider constraint sets with *precisely* two elements, the exceptions being the RCL 2000 specification language [2] and the work of Simon and Zurko [16].

It may be possible to enforce such constraints by entering the remaining set of permissions $\{p_1, p_2\}$ into ρ_u. If u now invokes permission p_2 (say), it is deleted from the blacklist, leaving a singleton set $\{p_3\}$. A subsequent request to invoke p_3 would then fail.

6. SUMMARY AND FUTURE WORK

We have developed a simple set-based specification scheme for authorization constraints in role-based access control systems. We have also suggested an enforcement model for a restricted subset of this scheme. To the author's knowledge, this is the first attempt at defining a specification and enforcement model for authorization constraints since the work by Simon and Zurko [16]. We believe that our specification is easier to understand than their scheme and that our enforcement model, which does not rely on maintaining a historical record of all previous system activity, is

likely to have lower performance overheads, particularly for large-scale applications.

There are several interesting directions for future work, some of which have been alluded to in the body of the paper. We would like to investigate whether constraint sets with an arbitrary number of elements can be enforced in a simple way. We would also like to find an intuitive scheme for defining different combinations of elements in the constraint scope and the constraint set in order to increase the range of constraints that our scheme can support.

The most ambitious goal is to develop abstract Java classes, `constraint` and `monitor`, that implement our constraint specification and enforcement schemes. The ultimate objective being to develop a generic middleware authorization constraint engine that can be instantiated by application developers and systems administrators to support enterprise-wide heterogeneous constraint authorization policies.

7. REFERENCES

[1] Abadi, M., and Fournet, C. Access control based on execution history. In *Proceedings of 10th Annual Network and Distributed System Security Symposium* (2003). To appear.

[2] Ahn, G.-J., and Sandhu, R. Role-based authorization constraints specification. *ACM Transactions on Information and System Security 3*, 4 (2000), 207–226.

[3] Bell, D., and LaPadula, L. Secure computer systems: Unified exposition and Multics interpretation. Tech. Rep. MTR-2997, Mitre Corporation, Bedford, Massachusetts, 1976.

[4] Bertino, E., Ferrari, E., and Atluri, V. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security 2*, 1 (1999), 65–104.

[5] Brewer, D., and Nash, M. The Chinese Wall security policy. In *Proceedings of 1989 IEEE Symposium on Security and Privacy* (Oakland, California, 1989), IEEE Computer Society Press, pp. 206–214.

[6] Clark, D., and Wilson, D. A comparison of commercial and military computer security policies. In *Proceedings of 1987 IEEE Symposium on Security and Privacy* (Oakland, California, 1987), pp. 184–194.

[7] Crampton, J., and Loizou, G. Structural complexity of conflict of interest policies. Tech. Rep. BBKCS-00-07, Birkbeck College, University of London, 2000.

[8] Edjlali, G., Acharya, A., and Chaudhary, V. History-based access control for mobile code. In *Proceedings of Fifth ACM Conference on Computer and Communications Security* (1998), pp. 38–48.

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

[10] Gavrila, S., and Barkley, J. Formal specification for role based access control user/role and role/role relationship management. In *Proceedings of Third ACM Workshop on Role-Based Access Control* (Fairfax, Virginia, 1998), pp. 81–90.

[11] Gligor, V., Gavrila, S., and Ferraiolo, D. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of 1998 IEEE Symposium on Research in Security and Privacy* (Oakland, California, 1998), pp. 172–183.

[12] Jaeger, T., and Tidswell, J. Practical safety in flexible access control models. *ACM Transactions on Information and System Security 4*, 2 (2001), 158–190.

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

[14] Sandhu, R. Transaction control expressions for separation of duties. In *Proceedings of 4th Aerospace Computer Security Conference* (Orlando, Florida, 1988), pp. 282–286.

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

[16] Simon, R., and Zurko, M. Separation of duty in role-based environments. In *Proceedings of 10th IEEE Computer Security Foundations Workshop* (Rockport, Massachusetts, 1997), pp. 183–194.