The *RSL99* Language for Role-Based Separation of Duty Constraints *

Gail-Joon Ahn and Ravi Sandhu
Laboratory for Information Security Technology
Information and Software Engineering Department
George Mason University
{gahn,sandhu}@isse.gmu.edu

Abstract

Separation of duty (SOD) is a fundamental technique for prevention of fraud and errors, known and practiced long before the existence of computers. It is discussed at several places in the literature, but there has been little work on specifying SOD policies in a systematic way. This paper describes a framework for specifying separation of duty and conflict of interest policies in role-based systems. To specify these policies, we need an appropriate language. We propose an intuitive formal language which uses system functions and sets as its basic elements. The semantics for this language is defined by its translation to a restricted form of first order predicate logic. We show how previously identified SOD properties can be expressed in our language. Moreover, we show there are other significant SOD properties which have not been previously identified in the literature. Unlike much of the previous work, this paper deals with SOD in the presence of role hierarchies. Our work shows that there are many alternate formulations of even the simplest SOD properties, with varying degree of flexibility and assurance. Our language provides us a rigorous foundation for systematic study of SOD properties.

*This work is partially supported by grants from the National Science Foundation and the National Security Agency at the Laboratory for Information Security Technology at George Mason University.

All correspondence should be addressed to Ravi Sandhu, ISE Department, Mail Stop 4A4, George Mason University, Fairfax, VA 22030, sandhu@isse.gmu.edu, www.list.gmu.edu.

1 INTRODUCTION

As a security principle, separation of duty (SOD) is a fundamental technique for prevention of fraud and errors, known and practiced long before the existence of computers. It is used to formulate multi-user control policies, requiring that two or more different users be responsible for the completion of a transaction or set of related transactions. The purpose of this principle is to minimize fraud by spreading the responsibility and authority for an action or task over multiple users, thereby raising the risk involved in committing a fraudulent act by requiring the involvement of more than one individual. A frequently used example is the process of preparing and approving purchase orders. If a single individual prepares and approves purchase orders, it is easy and tempting to prepare and approve a false order and pocket the money. If different users must prepare and approve orders, then committing fraud requires a conspiracy of at least two, which significantly raises the risk of disclosure and capture.

Although separation of duty is easy to motivate and understand intuitively, so far there is no formal basis for expressing this principle in computer security systems. Several definitions of SOD have been given in the literature. For the purpose of this paper we use the following definition.

> **Separation of duty** *reduces the possibility for fraud or significant errors (which can cause damage to an organization) by partitioning of tasks and associated privileges so cooperation of multiple users is required to complete sensitive tasks.*

Our objective in this paper is to study SOD in context of role-based access control (RBAC) [SCFY96]. RBAC has become a well-accepted and well-known approach for authorization and access control in modern systems.

We assume the reader is familiar with basic RBAC concepts and their underlying motivation. We have the following definition for interpreting SOD in role-based environments.

> **Role-Based separation of duty** *ensures SOD requirements in role-based systems by controlling membership in, activation of, and use of roles as well as permission assignment.*

There are several papers in the literature over the past decade which deal with separation of duty (as discussed in the next section). During this period various forms of SOD have been identified. Attempts have been made to systematically categorize these definitions. Notably, Simon and Zurko [SZ97] provide an informal characterization, and Gligor et al. [GGF98] provide a formalism of this characterization. However, this work has significant limitations. It omits important forms of SOD including session-based dynamic SOD needed for simulating lattice-based access control and Chinese Walls in RBAC [San93, San96]. It also does not deal with SOD in the presence of role hierarchies. Moreover, as will see, there are additional SOD properties that have not been identified in the previous literature.

In this paper we take a different approach to understand SOD. Rather than simply enumerating different kinds of SOD we propose a formal language for specifying different SOD properties. This language uses system functions and sets as its basic elements. It also has two non-deterministic functions which are introduced to replace explicit quantifiers. For ease of reference we call this language *RSL99* (Role-based Separation of duty Language 1999).

The rest of the paper is organized as follows. Section 2 reviews previous work on separation of duty. Section 3 defines the specification language *RSL99* including the basic elements and notation which are used to express separation of duty properties following by syntax and semantics of *RSL99* in section 4. Section 5 discusses how role-based SOD properties can be expressed in *RSL99*. Section 6 concludes the paper.

2 PRIOR WORK ON SOD

In this section, we discuss prior work on separation of duty which is a foundational principle in computer security.

Clark and Wilson [CW87] called attention to separation of duty as one of the major mechanisms to counter fraud and error while ensuring the correspondence between data objects within a system and the real world objects they represent. The Clark-Wilson scheme includes the requirement that the system maintain the separation of duty requirement expressed in the access control triples. It calls for certification by the security officer that these tuples provide adequate separation. It is therefore a static concept that is realized at design time.

Dynamic separation of duty provides greater flexibility by allowing a user to carry out conflicting operations, but only on distinct objects. Sandhu introduced notation for dynamic separation of duty in Transaction Control Expressions [San88]. Roles were used to specify who can issue which transaction steps. In Sandhu's model each user executing a step in a transaction had to be different. To enforce this, the history of the execution of each transaction sequence had to be maintained. The constraints specifying the roles that could execute each step were associated with an object. These constraints turned into the history specifying which user executed each step on that object. A weighted voting syntax allowed the specification of multiple person authorizations on a particular step on a particular object.

Baldwin [Bal90] introduced Named Protection Domains (NPDs) as a named, hierarchical grouping of database privileges and users. To help enforce separation of duty, a user could have only one of these NPDs activated at any time. The security administrator determined which NPDs could be activated, but there were no further restrictions on the graph of NPDs (other than it be acyclic). Thus, one activatable NPD could contain multiple activatable and non-activatable NPDs. While the activation restriction meant that a user could be in only one role (NPD) at a time, the security administrator could set up arbitrarily complex roles.

A number of new issues around separation of duty was raised by Nash and Poland's paper [NP90] of a portable security device used in the commercial world. Nash and Poland also proposed the notion of object based separation of duty, which forced every transaction against an object to be by a different user. They suggested using Sandhu's Transaction Control Expressions [San88] to maintain the history of an object's transactions.

Ferraiolo et al. [FCK95, FBK99] defined three kinds of separation of duty in their formal model of RBAC. The first two were static separation of duty and dynamic separation of duty. These variants were presented in previous work. The third kind was operational separation of duty which introduced the notion of a business function and the set of operations required for that function; a business function resembles the notion of task and task unit introduced by Thomas and Sandhu [TS94]. The formal definition of operational separation of duty stated that no role can contain the permissions to execute all of the operations necessary to a single business function. This forces all business functions to require at least two roles to be used for their completion. The informal description of operational separation of duty assumes the roles involved have disjoint memberships (static separation of duty), so that no single person has access to all the operations in a business function.

Kuhn's paper [Kuh97] focussed on the time when exclusion is introduced, and the degree to which two roles conflict. As far as time is concerned, mutual exclusion can be defined at role authorization time, or at run time. As he observed, these correspond to static and dynamic separation of duties respectively. Kuhn also distinguished between complete and partial mutual exclusion of roles.

Simon and Zurko [SZ97] enumerated different kinds of separation of duty such as static separation of duty (or strong exclusion), dynamic separation of duty (or weak exclusion), and object-based separation of duty. Simon and Zurko's enumeration of kinds of conflict of interest also includes four more kinds which all have to do with complex tasks involving several interrelated steps, say in a workflow management system. They tried to enumerate all the variations of SOD that have been called out in one source or another but their description of SOD was informal.

Gligor et al. [GGF98] enumerated several forms of SOD properties using first order predicate logic which causes difficulties to understand the properties. They missed the important SOD properties in RBAC such as session-based SOD and SOD in role hierarchies. To constrain sessions, we should consider each session of a set of sessions as well as a set of sessions. Also, SOD should be applied with the role hierarchies.

Nyanchama and Osborn [NO99] discussed a taxonomy of conflict of interest types such as user-user conflicts and privilege-privilege conflicts. They also discussed such conflicts in great detail with respect to their role graph model.

3 ROLE-BASED SOD LANGUAGE (*RSL99*)

In this section we define a new formal language for specifying SOD properties in role-based systems. The language is described here informally and intuitively. Formal syntax and semantics are given in next section. To develop this language, we need a model for role-based systems. A general RBAC model, commonly called RBAC96 [SCFY96, San97] has become a widely cited reference in this arena. For the most part, *RSL99* components are built upon RBAC96. This model is illustrated in the top part of figure 1.

Our work also builds upon SOD properties analyzed in [SZ97] and formalized in [GGF98]. Even though their work is state-of-art there are several shortcomings. As we address those shortcomings, we introduce the motivation for our work. Their work does not have the notion of role hierarchies. Also, it misses the concept of session-based SOD which deals with SOD property in a single session. This form of dynamic SOD is useful for simulating lattice-based access control and Chinese Walls in RBAC [San93, San96]. Conflicting users and privileges are also not dealt with. From these observations, we are led to identify other significant SOD properties which have not been previously identified in the literature.

In this section we introduce the basic elements on which *RSL99* is based and introduce notation and definitions that will be used in the rest of this paper. Also, we introduce the basic constructs of our specification language *RSL99*. We will show in subsequent sections how these constructs can be used to specify the various separation of duty properties.

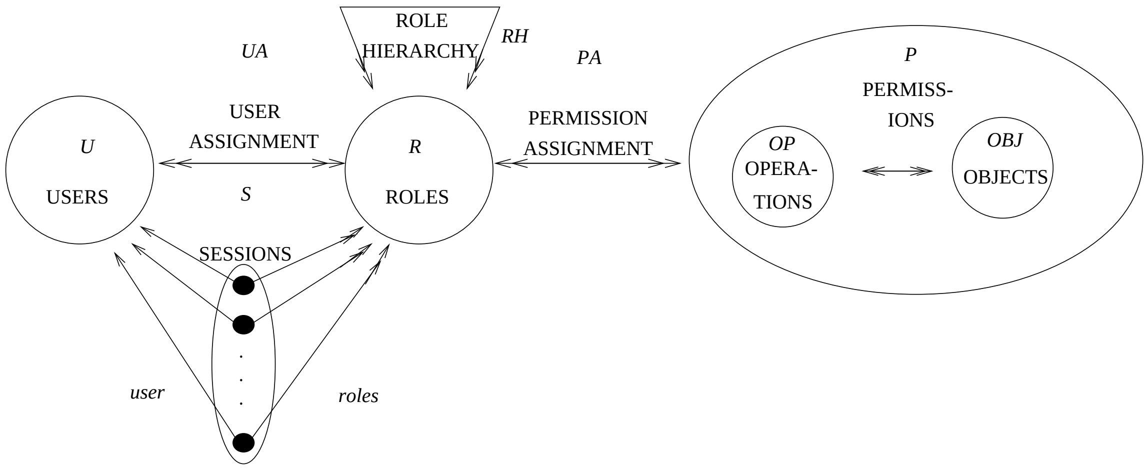
The rest of this section is organized as follows. Section 3.1 introduces the basic elements and system functions which are from RBAC96 model. Section 3.2 introduce the basic elements and non-deterministic system functions which are newly proposed functions in this work. These non-deterministic functions are the core concept of this work. They eliminate use of explicit quantifiers resulting in an intuitive language.

3.1 Basic Elements and System Functions: from RBAC96

The basic elements on which *RSL99* is based and system functions that will be used in the rest of this paper are defined in figure 1. Figure 1 also shows the RBAC96 model which is the context for these definitions.

RSL99 has six entity sets called users (`U`), roles (`R`), objects (`OBJ`), operations (`OP`), permissions (`P`), and sessions (`S`). These are interpreted as in RBAC96. A user is a human being. A role is a named job function within the organization that describes the authority and responsibility conferred on a member of the role. Objects are passive entities that contain or receive information. An operation is an executable image of a program, which upon execution causes information flow between objects. A permission is an approval of a particular mode of operation to one or more objects in the system.

A session is a mapping between a user and an activated subset of the set of roles the user is assigned to. The function `user` gives us the user associated with a



- $\mathtt{U}$ = a set of users, $\{u_1, ..., u_n\}$; $\mathtt{R}$= a set of roles, $\{r_1, ..., r_n\}$;
- $\mathtt{OP}$ = a set of operations, $\{op_1, ..., op_n\}$; $\mathtt{OBJ}$ = a set of objects, $\{obj_1, ..., obj_n\}$;
- $\mathtt{P} = \mathtt{OP} \times \mathtt{OBJ}$, a set of permissions, $\{p_1, ..., p_n\}$; and
- $\mathtt{S}$ = a set of sessions, $\{s_1, ..., s_n\}$.
  - $\mathtt{user} : \mathtt{S} \to \mathtt{U}$, a function mapping each session s_i to the single user.
  - $\mathtt{roles} : \mathtt{S} \to 2^{\mathtt{R}}$, a function mapping the set $\mathtt{S}$ to a set of roles $\mathtt{R}$.
    $\mathtt{roles}(s_i) \subseteq \{r \in \mathtt{R} \mid (\mathtt{user}\ (s_i), r) \in \mathtt{UA}\}$
- $\mathtt{RH} \subseteq \mathtt{R} \times \mathtt{R}$ is a partial order on $\mathtt{R}$ called the role hierarchy or role dominance relation, written as $\preceq$.
- $\mathtt{UA} \subseteq \mathtt{U} \times \mathtt{R}$, a many-to-many user-to-role assignment relation.
- $\mathtt{PA} \subseteq \mathtt{P} \times \mathtt{R} = \mathtt{OP} \times \mathtt{OBJ} \times \mathtt{R}$, a many-to-many permission-to-role assignment relation.
- $\mathtt{user} : \mathtt{R} \to 2^{\mathtt{U}}$, a function mapping each role r_i to a set of users.
  $\mathtt{user}(r_i) = \{u \in \mathtt{U} \mid (u, r_i) \in \mathtt{UA}\}$
- $\mathtt{roles} : \mathtt{U} \cup \mathtt{P} \to 2^{\mathtt{R}}$, a function mapping the set $\mathtt{U}$ and $\mathtt{P}$ to a set of roles $\mathtt{R}$.
  $\mathtt{roles}^* : \mathtt{U} \cup \mathtt{P} \cup \mathtt{S} \to 2^{\mathtt{R}}$, extends $\mathtt{roles}$ in presence of role hierarchy
  $\mathtt{roles}(u_i) = \{r \in \mathtt{R} \mid (u_i, r) \in \mathtt{UA}\}$ $\quad$ $\mathtt{roles}^*(u_i) = \{r \in \mathtt{R} \mid (\exists\, r' \succeq r)[(u_i, r') \in \mathtt{UA}]\}$
  $\mathtt{roles}(p_i) = \{r \in \mathtt{R} \mid (p_i, r) \in \mathtt{PA}\}$ $\quad$ $\mathtt{roles}^*(p_i) = \{r \in \mathtt{R} \mid (\exists\, r' \preceq r)[(p_i, r') \in \mathtt{PA}]\}$
  $\mathtt{roles}(s_i)$ = defined above $\quad$ $\mathtt{roles}^*(s_i) = \{r \in \mathtt{R} \mid (\exists\, r' \succeq r)[r' \in \mathtt{roles}(s_i)]\}$
- $\mathtt{sessions} : \mathtt{U} \to 2^{\mathtt{S}}$, a function mapping each user u_i to a set of sessions.
- $\mathtt{permissions} : \mathtt{R} \to 2^{\mathtt{P}}$, a function mapping each role r_i to a set of permissions.
  $\mathtt{permissions}^* : \mathtt{R} \to 2^{\mathtt{P}}$, extends $\mathtt{permissions}$ in presence of role hierarchy
  $\mathtt{permissions}(r_i) = \{p \in \mathtt{P} \mid (p, r_i) \in \mathtt{PA}\}$ $\quad$ $\mathtt{permissions}^*(r_i) = \{p \in \mathtt{P} \mid (\exists\, r \preceq r_i)[(p, r_i) \in \mathtt{PA}]\}$
- $\mathtt{operations} : \mathtt{R} \times \mathtt{OBJ} \to 2^{\mathtt{OP}}$, a function mapping each role r_i and object obj_i to a set of operations.
  $\mathtt{operations}(r_i, obj_i) = \{op \in \mathtt{OP} \mid (op, obj_i, r_i) \in \mathtt{PA}\}$

Figure 1: Basic Elements and System Functions : from RBAC96 Model

session and roles gives us the roles activated in a session. Both functions do not change during the life of a session. (This is a slight simplification from RBAC96 which does allow roles in a session to change.)

Hierarchies are a natural means for structuring roles to reflect an organization's lines of authority and responsibility. Mathematically, these hierarchies are partial orders. A partial order is a reflexive, transitive, and antisymmetric relation, so that if $y \prec x$ then role x inherits the permissions of role y, but not vice versa.

The user assignment relation UA is a many-to-many relation between users and roles. Similarly the permission-assignment relation PA is a many-to-many relation between permissions and roles. Users are authorized to use the permissions of roles to which they are assigned. This is the essence of RBAC.

The remaining functions defined in figure 1 are built from the sets, relations, and functions discussed above. In particular note that user and roles can have different types of arguments so we are overloading these symbols. Also the definition of $\texttt{roles}^*$ is carefully formulated to reflect the role inheritance with respect to users and sessions going downward and with respect to permissions going upward. In other words a permission in a junior role is available to senior roles, and activation of a senior role makes available permissions of junior roles. This is well-accepted concept in the RBAC literature. Using a single symbol $\texttt{roles}^*$ simplifies our notation so long as we keep this duality of inheritance in mind.

In *RSL99*, for simplicity, we assume that role hierarchies, UA, and PA do not change. We consider one snapshot in a system at a time and SOD properties are applied only to that snapshot.

3.2 Basic Elements and Non-deterministic Functions: beyond RBAC96

Additional elements and system functions used in *RSL99* are defined in figure 2. For mutually disjoint organizational roles such as those of purchasing manager and accounts payable manager, the same individual is generally not permitted to belong to both roles. We defined these mutually disjoint roles as conflicting roles. We assume that there is a collection CR of sets of roles which have been defined to conflict.

The concept of conflicting permissions defines conflict in terms of permissions rather than roles. Thus the permission to issue purchase orders and the permission to issue payments are conflicting, irrespective of the roles to which they are assigned. We denote sets of conflicting permissions as CP. As we will see defining conflict in terms of permissions offers greater assurance than defining it in terms of roles. Conflict defined in terms of roles allows conflicting permissions to be assigned to the same role by error (or malice). Conflict defined in terms of permissions eliminates this possibility.

In the real world, conflicting users should be also considered. For example, for the process of preparing and approving purchase orders, it might be company policy that members of the same family should not prepare the purchase order, and also be a user who approves that order. This kind of SOD property is not discussed in [GGF98, SZ97]. The concept is mentioned in [NO99] but in a general way without identification of specific properties in this class. We denote sets of conflicting users as CU.

RSL99 has two non deterministic functions, oneelement and allother (first introduced by Chen and Sandhu [CS95]). These are introduced to replace explicit quantifiers. The elimination of explicit quantifiers from our language keeps it simple and intuitive. The oneelement(X) function allows us to get one element x_i from set X. We usually write oneelement as OE. Multiple occurrence of OE(X) in a single *RSL99* statement all select the same element x_i from X. With allother(X) we can get a set by taking out one element. We usually write allother as AO.

These two non-deterministic functions are related by context, because for any set S, $\{\texttt{OE}(S)\} \cup \texttt{AO}(S) = S$, and at the same time, neither is a deterministic function. In order to illustrate how to use these two functions to specify SOD properties, we take the requirement of static separation of duty property which is the simplest variation of SOD. For the moment assume there is no role hierarchy.

> Requirement: *No user should be assigned to two conflicting roles.* In other words, conflicting roles can not have common users. We can express this requirement as below.
> Expression: $\texttt{OE}(\texttt{OE}(\texttt{CR})) \in \texttt{roles}(\texttt{OE}(\texttt{U})) \Longrightarrow \texttt{AO}(\texttt{OE}(\texttt{CR})) \cap \texttt{roles}(\texttt{OE}(\texttt{U})) = \phi$

OE(OE(CR)) means a conflicting role which is an element of set OE(CR) and roles(OE(U)) returns all roles which are assigned to a single user OE(U). AO(OE(CR)) means conflicting roles excluding one role which is from OE(OE(CR)). We can interpret the above expression as saying that if a user has been assigned to one conflicting role, that user cannot be assigned to any other conflicting role. We can also specify this property in many different ways using *RSL99* such as $|\texttt{roles}(\texttt{OE}(\texttt{U})) \cap \texttt{OE}(\texttt{CR})\ | \leq 1$ or $\texttt{user}(\texttt{OE}(\texttt{OE}(\texttt{CR}))) \cap \texttt{user}(\texttt{AO}(\texttt{OE}(\texttt{CR}))) = \phi$

RSL99 system functions do not include a time or

- $\mathtt{CR}$ = a collection of conflicting role sets, $\{cr_1, ..., cr_n\}$, where $cr_i = \{r_i, ..., r_k\} \subseteq \mathtt{R}$
- $\mathtt{CP}$ = all conflicting permission sets, $\{cp_1, ..., cp_n\}$, where $cp_i = \{p_i, ..., p_k\} \subseteq \mathtt{P}$
- $\mathtt{CU}$ = all conflicting user sets, $\{cu_1, ..., cu_n\}$, where $cu_i = \{u_i, ..., u_k\} \subseteq \mathtt{U}$
- $\mathtt{oneelement}(\mathrm{X}) = x_i$, where $x_i \in \mathrm{X}$
- $\mathtt{allother}(\mathrm{X}) = \mathrm{X}$ - $\{\mathtt{OE}(\mathrm{X})\}$

Figure 2: Basic Elements and Non-deterministic Functions: beyond RBAC96 Model

state variable in their structure. So we assume that each function considers the current time or state. For example, if we use `sessions` function in the expression, this function maps a user u_i to a set of current sessions which are established by user u_i.

As a general notational device we have the following convention.

- For any set valued function *f* defined on set X, We understand $f(\mathrm{X}) = f(x_1) \cup f(x_2) \cup ... \cup f(x_n)$, where $\mathrm{X}=\{x_1, x_2, x_3, ..., x_n\}$.

4 FORMAL SYNTAX AND SEMANTICS OF *RSL99*

We now give the formal syntax and semantic of *RSL99*. For the syntax we use usual BNF. For semantics of *RSL99* we identify a restricted form of first order predicate logic which corresponds exactly to *RSL99*.

4.1 The Syntax

The syntax of *RSL99* is defined by the syntax diagram and grammar given in figure 3. The rules take the form of flow diagrams. The possible paths represent the possible sequence of symbols. Backus Normal Form (BNF) is also used to describe the grammar of *RSL99* as shown in the bottom of figure 3. The symbols of this form are: ::= meaning "is defined as" and | meaning "or". Also we denote `oneelement` and `allother` as `OE` and `AO` respectively. We assume that the type of arguments of functions should follow the function descriptions presented in section 3.

4.2 Formal Semantics

Next, we discuss the formal semantics for *RSL99*. Any property written in *RSL99*, called *RSL99* expression, can be translated to an expression which is written in a restricted form of first order predicate logic which we call RFOPL. The syntax of RFOPL is described at the end of this section. The translation algorithm we developed, namely *Reduction*, converts a *RSL99* expression to an RFOPL expression. This algorithm is outlined in figure 4(a). *Reduction* algorithm eliminates `AO` function(s) from *RSL99* expression in the first step. Then we translate `OE` terms iteratively into an element introducing universal quantifiers from left to right. If we have nested `OE` functions in *RSL99* expression, translation will be started from innermost `OE` terms. The analysis of the running time depends on the number of `OE` term. Therefore, this algorithm can translate *RSL99* expression to RFOPL expression in time $\mathcal{O}$(n) supposing that the number of `OE` term is n.

For example, the following *RSL99* expression can be converted to RFOPL expression according to the sequences below.

- *RSL99* expression:

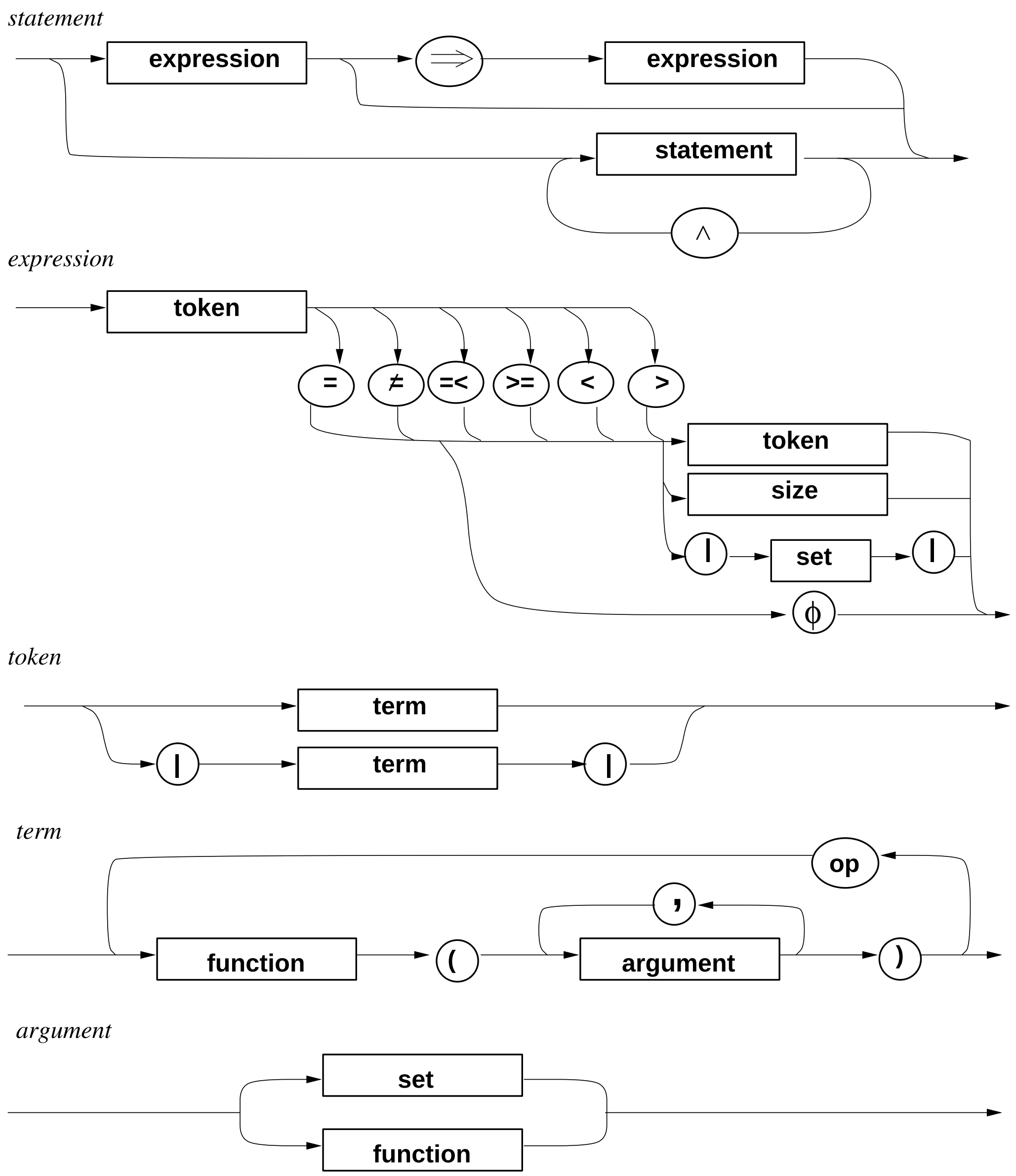
$$\mathtt{OE}(\mathtt{OE}(\mathtt{CR})) \in \mathtt{roles}(\mathtt{OE}(\mathtt{U})) \Longrightarrow \mathtt{AO}(\mathtt{OE}(\mathtt{CR})) \cap \mathtt{roles}(\mathtt{OE}(\mathtt{U})) = \phi$$

- RFOPL expression:

1. $\mathtt{OE}(\mathtt{OE}(\mathtt{CR})) \in \mathtt{roles}(\mathtt{OE}(\mathtt{U})) \Longrightarrow (\mathtt{OE}(\mathtt{CR}) - \{\mathtt{OE}(\mathtt{OE}(\mathtt{CR}))\}) \cap \mathtt{roles}(\mathtt{OE}(\mathtt{U})) = \phi$
2. $\forall\, cr \in \mathtt{CR}$: $\mathtt{OE}(cr) \in \mathtt{roles}(\mathtt{OE}(\mathtt{U})) \Longrightarrow (cr - \{\mathtt{OE}(cr)\}) \cap \mathtt{roles}(\mathtt{OE}(\mathtt{U})) = \phi$
3. $\forall\, cr \in \mathtt{CR}, \forall\, r \in cr$: $r \in \mathtt{roles}(\mathtt{OE}(\mathtt{U})) \Longrightarrow (cr - \{r\}) \cap \mathtt{roles}(\mathtt{OE}(\mathtt{U})) = \phi$
4. $\forall\, cr \in \mathtt{CR}, \forall\, r \in cr, \forall\, u \in \mathtt{U}$: $r \in \mathtt{roles}(u) \Longrightarrow (cr - \{r\}) \cap \mathtt{roles}(u) = \phi$

The resulting RFOPL expression will have the following general structure.

1. The RFOPL expression has a (possibly empty) sequence of universal quantifiers as a left prefix, and these are the only quantifiers it can have. We call this sequence the *quantifier part*.
2. The quantifier part will be followed by a predicate separated by colon (:), i.e., *universal quantifier part* **:** *predicate*



op ::= $\in$ | $\cap$ | $\cup$ | $-$

$size$ ::= ϕ | 1 | ... | N

set ::= `U` | `R` | `OP` | `OBJ` | `P` | `S` | `CU` | `CR` | `CP`

$function$ ::= `user` | `roles` | `roles`* | `sessions` | `permissions` |
`permissions`* | `operations` | `OE` | `AO`

Figure 3: Syntax of Language

Reduction **Algorithm**
Input: *RSL99* expression ; Output: RFOPL expression

Let `Simple-OE` term be either `OE`(*set*), or `OE`(**`function`**(*element*)).
Let *set* be an element of set-name set, **`function`** be an element of function-name set.

- set-name $=\{$`U`, `R`, `OP`, `OBJ`, `P`, `S`, `CR`, `CU`, `CP`, `T`, `HU`, `HS`, cr, cu, cp$\}$
- function-name $=\{$**`user`**, **`roles`**, **`roles`***, **`sessions`**, **`permissions`**, **`permissions`***, **`operations`**$\}$

1. `AO` elimination
   replace all occurrences of `AO`(*expr*) with (*expr* - $\{$`OE`(*expr*)$\}$);

2. `OE` elimination
   While There exists `Simple-OE` term in *RSL99* expression
      choose `Simple-OE` term;
      call **`reduction`** procedure;
   End

   Procedure **`reduction`**
      case (i) `Simple-OE` term is `OE`(*set*)
       create new value x;
       put $\forall\, x \in$ *set* to right of existing quantifier(s);
       replace all occurrences of `OE`(*set*) by x;

      case (ii) `Simple-OE` term is `OE`(**`function`**(*element*))
       create new value x;
       put $\forall\, x \in$ **`function`**(*element*) to right of existing quantifier(s);
       replace all occurrences of `OE`(**`function`**(*element*)) by x;
   End

(a) Reduction Algorithm

Construction **Algorithm**
Input: RFOPL expression ; Output: *RSL99* expression

1. Construction *RSL99* expression from RFOPL expression
   While There exists quantifier in RFOPL expression
      choose the rightmost quantifier $\forall\, x \in$ X;
      pick values x and X from the chosen quantifier;
      replace all occurrences of x by `OE`(X);
   End

3. Replacement of `AO`
   if there is (*expr* - $\{$`OE`(*expr*)$\}$) in RFOPL expression
    replace it with `AO`(*expr*);

(b) Construction Algorithm

Figure 4: Reduction and Construction

3. The predicate has no free variables or constant symbols. All variables are declared in the quantifier part, e.g., $\forall\, r \in \mathtt{R},\ \forall\, u \in \mathtt{U} :\ r \in \mathtt{roles}(u)$.

4. The order of quantifiers is determined by the sequence of OE elimination. In some cases this order is important so as to reflect the nesting of OE terms in the *RSL99* expression. For example, in $\forall\, cr \in \mathtt{CR},\ \forall\, r \in cr, \forall\, u \in \mathtt{U} :$ *predicate*; the set cr which is used in the second quantifier must be declared in a previous quantifier as an element, such as cr in the first quantifier.

5. *predicate* follows most of rules in the syntax of *RSL99* except *term* syntax in figure 3. Figure 5 shows the syntax which *predicate* should follow to express *term*. In figure 5, element means that any element which belongs to *set* described in figure 3.

The above discussion defines the syntax of RFOPL. A complete formal definition can be given from these observation and is omitted for simplicity.

4.3 Soundness and Completness

Next we discuss the algorithm *Construction* that constructs a *RSL99* expression from a RFOPL expression which has the syntax given above. The algorithm is described in figure 4(b). Firstly, this algorithm repeatedly chooses the rightmost quantifier in RFOPL expression and constructs the corresponding OE term by eliminating the variable of that quantifier. After all quantifiers are eliminated the algorithm constructs AO terms according to the formal definition of AO function. The running time of algorithm obviously depends on the number of quantifiers in RFOPL expression.

For example, the following RFOPL expression can be converted to *RSL99* expression according to the sequences described below.

•RFOPL expression:

$\forall\, cr \in \mathtt{CR},\ \forall\, r \in cr,\ \forall\, u \in \mathtt{U},\ \forall\, s \in \mathtt{sessions}(u) :\ r \in \mathtt{roles}(s) \Longrightarrow (cr - \{r\}) \cap \mathtt{roles}(s) = \phi$

•*RSL99* expression :

1. $\forall\, cr \in \mathtt{CR}, \forall\, r \in cr, \forall\, u \in \mathtt{U}$: $r \in \mathtt{roles}(\mathtt{OE}(\mathtt{sessions}(u)) \Longrightarrow (cr - \{r\}) \cap \mathtt{roles}(\mathtt{OE}(\mathtt{sessions}(u))=\phi$

2. $\forall\, cr \in \mathtt{CR}, \forall\, r \in cr$: $r \in \mathtt{roles}(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U})))) \Longrightarrow (cr - \{r\}) \cap \mathtt{roles}(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U}))))=\phi$

3. $\forall\, cr \in \mathtt{CR}$: $\mathtt{OE}(cr) \in \mathtt{roles}(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U})))) \Longrightarrow (cr - \{\mathtt{OE}(cr)\}) \cap \mathtt{roles}(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U}))))=\phi$

4. $\mathtt{OE}(\mathtt{OE}(\mathtt{CR})) \in \mathtt{roles}(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U})))) \Longrightarrow (\mathtt{OE}(\mathtt{CR})\text{-}\{\mathtt{OE}(\mathtt{OE}(\mathtt{CR}))\}) \cap \mathtt{roles}(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U})))) = \phi$

5. $\mathtt{OE}(\mathtt{OE}(\mathtt{CR})) \in \mathtt{roles}(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U})))) \Longrightarrow \mathtt{AO}(\mathtt{OE}(\mathtt{CR})) \cap \mathtt{roles}(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U}))))=\phi$

We introduced two algorithms, namely *Reduction* and *Construction*, that can reduce and construct *RSL99* expression. We now introduce theorems regarding translation between *RSL99* and RFOPL expressions. Let $\mathcal{R}(expr)$ denote the RFOPL expression translated by *Reduction* algorithm. Let $\mathcal{C}(expr)$ denote the *RSL99* expression constructed by *Construction* algorithm. The relationship between *RSL99*-RFOPL translations above is represented by the following theorems.

Theorem 1 *Given RSL99 expression α, α can be translated into* RFOPL *expression β. Also α can be reconstructed from β.*

$\mathcal{C}(\mathcal{R}(\alpha)) = \alpha$

Proof Sketch: It suffices to consider the case where α has no AO in it. If α has no OE in it, this theorem is obviously correct. Given α, we reduce the OE term from it, replacing it with a variable. During this replacement, we also put quantifiers from left to right in β. The construction algorithm reverses this translation from right to left. A formal inductive proof based on this argument can be given. □

Theorem 2 *Given* RFOPL *expression β, β can be translated into RSL99 expression α. Also β' which is logically equivalent to β can be reconstructed from α.*

$\mathcal{R}(\mathcal{C}(\beta)) = \beta'$

Proof Sketch: In case β does not have any quantifier in it, this theorem is obviously true. Otherwise, during construction of α from β, we choose the rightmost quantifier in β and replace the element of that quantifier with OE term. Given the constructed expression α, we reduce the OE term from it, replacing it with a variable. This variable might be different from β because **While-End** loop in reduction algorithm has non-deterministic choice for reduction of OE term. And the arrangement of quantifiers might also be different from β but it does not affect the structure of RFOPL expression β' because the reduction of OE term is always from the inner most one first. Finally we can get β' which is logically equivalent to β. A formal inductive proof based on this argument can be given. □

Theorem 1 establishes soundness of *RSL99* and theorem 2 establishes its completeness relation to RFOPL.

5 SOD PROPERTIES

SOD is a well-known principle for preventing fraud by identifying conflicting roles—such as Purchasing Manager and Accounts Payable Manager—and ensuring

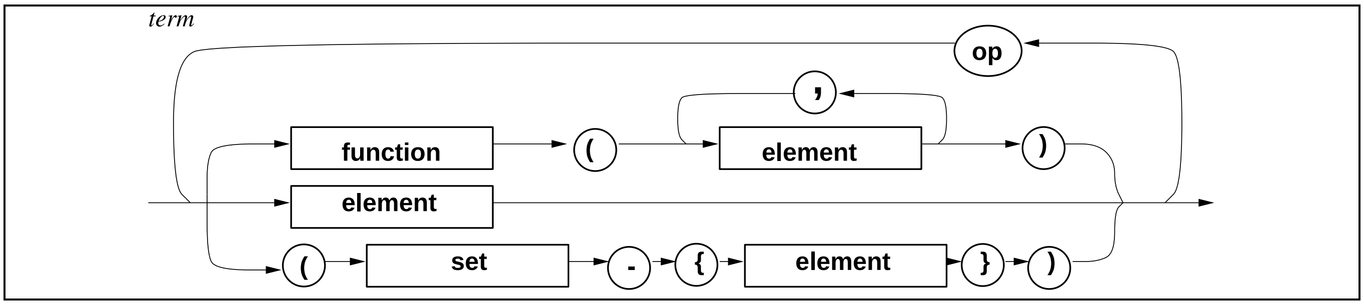


Figure 5: Syntax of restricted FOPL expression

that the same individual can belong to at most one conflicting role. Static SOD applies to the user-assignment relation and dynamic SOD applies to the activated roles in session(s). In this section, we show how *RSL99* can be used to specify the various separation of duty properties.

5.1 Static SOD

Static SOD (SSOD) is the simplest variation of SOD. In table 1 we show our expression of several forms of SSOD. These include new forms of SSOD which have not previously been identified in the literature. This demonstrates how *RSL99* helps us in understanding SOD and discovering new basic form of it.

Property 1 is the most straightforward property. The SSOD requirement is that no user should be assigned to two roles which are conflicting each other. In other words, it means that conflicting roles cannot have common users. *RSL99* can clearly express this property. This property is the classic formulation of SSOD which is identified by several papers including [GGF98, Kuh97, SCFY96]. It is a role-centric property.

Property 2 follows the same intuition as property 1, but is permission-centric. Property 2 says that a user can have at most one conflicting permission acquired through roles assigned to the user. Property 2 is a stronger formulation than property 1 which prevents mistakes in role-permission assignment. This kind of property has not been previously mentioned in the literature. *RSL99* helps us discover such omissions in previous work. In retrospect property 2 is an "obvious property" but there is no mention of this property in over a decade of SOD literature. Even though property 2 allows more flexibility in role-permission assignment since the conflicting roles are not predefined, it can also generate roles which cannot be used at all. For example, two conflicting permissions can be assigned to a role. Property 2 simply requires that no user can be assigned to such a role or any role senior to it, which makes that role quite useless. Thus property 2 prevents certain kinds of mistakes in role-permissions but tolerates others.

Property 3 eliminates the possibility of useless roles with an extra condition, $|\texttt{permissions}^*(\texttt{OE}(\texttt{R})) \cap \texttt{OE}(\texttt{CP})| \leq 1$. This condition ensures that each role can have at most one conflicting permission without consideration of user-role assignment.

With this new condition, we can extend property 1 in presence of conflicting permissions as property 4. In property 4 we have another additional condition that conflicting permissions can only be assigned to conflicting roles. In other words, non-conflicting roles cannot have conflicting permissions. The net effect is that a user can have one conflicting permissions via roles assigned to the user.

Property 4 can be viewed as a reformulation of property 3 in a role-centric manner. Property 3 does not stipulate a concept of conflicting roles. However, we can interpret conflicting roles to be those that happen to have conflicting permissions assigned to them. Thus for every cp_i we can define $cr_i = \{r \in \texttt{R} \mid cp_i \cap \texttt{permissions}(r) \neq \phi\}$. With this interpretation, properties 2 and 4 are essentially identical. The viewpoint of property 3 is that conflicting permissions get assigned to distinct roles which thereby become conflicting, and therefore cannot assigned to the same user. Which roles are deemed conflicting is not determined a priori but is a side-effect of permission-role assignment. The viewpoint of property 4 is that conflicting roles are designated in advance and conflicting permissions must be restricted to conflicting roles. These properties have different consequences on how roles get designed and managed but essentially achieve the same objective with respect to separation of conflicting permissions. Both properties achieve this goal with much higher assurance than property 1. Property 2 achieves

Properties	Expressions
1. SSOD-CR	$\mid \texttt{roles}^*(\texttt{OE(U)}) \cap \texttt{OE(CR)} \mid \leq 1$
2. SSOD-CP	$\mid \texttt{permissions}(\texttt{roles}^*(\texttt{OE(U)})) \cap \texttt{OE(CP)} \mid \leq 1$
3. Variation of 2	$(2) \wedge \mid \texttt{permissions}^*(\texttt{OE(R)}) \cap \texttt{OE(CP)} \mid \leq 1$
4. Variation of 1	$(1) \wedge \mid \texttt{permissions}^*(\texttt{OE(R)}) \cap \texttt{OE(CP)} \mid \leq 1$ $\wedge\ \texttt{permissions}(\texttt{OE(R)}) \cap \texttt{OE(CP)} \neq \phi \Longrightarrow \texttt{OE(R)} \cap \texttt{OE(CR)} \neq \phi$
5. SSOD-CU	$(1) \wedge \mid \texttt{user}(\texttt{OE(CR)}) \cap \texttt{OE(CU)} \mid \leq 1$
6. Yet another variation	$(4) \wedge (5)$

Table 1: Static Separation of Duty

this goal with similar high assurance but allows for the possibility of useless roles. Thus, even in the simple situation of static SOD, we have a number of alternatives offering different degrees of assurance and flexibility.

Property 5 is a very different property and is also new to the literature. With a notion of conflicting users, we identify new forms of SSOD. Property 5 says that two conflicting users cannot be assigned to roles in the same conflicting role set. This property is useful because it is much easier to commit fraud if two conflicting users can have different conflicting roles in the same conflicting role set. This property prevents this kind of situation in role-based systems. A collection of conflicting users is less trustworthy than a collection of non-conflicting users, and therefore should not be mixed up in the same conflicting role set. This property has not been previously identified in the literature.

We also identify a composite property which includes conflicting users, roles and permissions. Property 6 combines property 4 and 5 so that conflicting users cannot have conflicting roles from the same conflict set while assuring that conflicting roles have at most one conflicting permission from each conflicting permission sets. This property supports SSOD in user-role and role-permission assignment with respect to conflicting users, roles, and permissions.

5.2 Dynamic SOD

In RBAC systems, a dynamic SOD (DSOD) property with respect to the roles activated by the users requires that no user can activate two conflicting roles. In other words, conflicting roles may have common users but users can not simultaneously activate roles which are conflicting each other. From this requirement we can express user-based Dynamic SOD as property 1. We can also identify a Session-based Dynamic SOD property which can apply to the single session as property 2. We can also consider these properties with conflicting users such as property 1-1 and 2-1. Additional analysis of dynamic SOD properties based on conflicting permissions can also be pursued as was done for static SOD.

5.3 History-based SOD

The concept history-based SOD has also been identified in the literature [GGF98]. With suitable extensions to handle time and history, *RSL99* can specify these properties. Space limitations preclude further discussion of this issue here.

6 CONCLUSION

In this paper we have introduced an intuitive formal language *RSL99* which allows us to systematically study SOD properties in role-based systems. We have given a formal syntax and semantics for *RSL99* and have demonstrated its soundness and completeness. We have shown that *RSL99* allows us to investigate nuances of static SOD and dynamic SOD in a way that has not been possible so far. This has led to formulation of static SOD properties that have not identified in a decade of literature on SOD.

We have barely begun exploring SOD properties by means of *RSL99*. In future work we expect to use *RSL99* to advance our understanding of SOD in diverse forms at a rapid pace.

References

[Bal90] Robert W. Baldwin. Naming and grouping privileges to simplify security management in large database. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 61–70, Oakland, CA, April 1990.

[CS95] Fang Chen and Ravi Sandhu. Constraints for role based access control. In *Proceedings of 1st ACM Workshop on Role-Based*

Properties	Expressions
1. User-based DSOD	$\mid \texttt{roles}^*(\texttt{sessions}(\texttt{OE}(\texttt{U}))) \cap \texttt{OE}(\texttt{CR}) \mid \leq 1$
1-1. User-based DSOD with CU	$\mid \texttt{roles}^*(\texttt{sessions}(\texttt{OE}(\texttt{OE}(\texttt{CU})))) \cap \texttt{OE}(\texttt{CR}) \mid \leq 1$
2. Session-based DSOD	$\mid \texttt{roles}^*(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U})))) \cap \texttt{OE}(\texttt{CR}) \mid \leq 1$
2-1. Session-based DSOD with CU	$\mid \texttt{roles}^*(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{OE}(\texttt{CU}))))) \cap \texttt{OE}(\texttt{CR}) \mid \leq 1$

Table 2: Dynamic Separation of Duty

Access Control, pages 39–46, Gaithersburg, MD, November 1995.

[CW87] D. D. Clark and D. R. Wilson. A comparision of commercial and military computer security plocies. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 184–194, April 1987.

[FBK99] David F. Ferraiolo, John F. Barkley, and D. Richard Kuhn. A role based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and Systems Security*, 2(1):34–64, February 1999.

[FCK95] David Ferraiolo, Janet Cugini, and Richard Kuhn. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 241–48, New Orleans, LA, December 11-15 1995.

[GGF98] Virgil D. Gligor, Serban I. Gavrila, and David 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, Oakland, CA, May 1998.

[Kuh97] D. Richard Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control*, Fairfax, VA, October 1997.

[NO99] Matunda Nyanchama and Sylvia Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and Systems Security*, 2(1):3–33, February 1999.

[NP90] M.N. Nash and K.R. Poland. Some conundrums concerning separation of duty. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 201–207, Oakland, CA, May 1990.

[San88] Ravi Sandhu. Transaction control expressions for separation of duties. In *Proceedings of 4th Aerospace Computer Security Conference*, pages 282–286, Orlando, FL, December 1988.

[San93] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, November 1993.

[San96] Ravi S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In Elisa Bertino, editor, *Proc. Fourth European Symposium on Research in Computer Security*. Springer-Verlag, Rome, Italy, 1996. Published as *Lecture Notes in Computer Science, Computer Security–ESORICS96.*

[San97] Ravi Sandhu. Rationale for the RBAC96 family of access control models. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*. ACM, 1997.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[SZ97] R. T. Simon and M. E. Zurko. Separation of duty in role-based environments. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 183–194, Rockport, MA, December 1997.

[TS94] Roshan Thomas and Ravi S. Sandhu. Conceptual foundations for a model of task-based authorizations. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 66–79, Franconia, NH, June 1994.