On the Formal Definition of Separation-of-Duty Policies and their Composition

Virgil D. Gligor
Electrical Engineering Department
University of Maryland
College Park, MD. 20742

Serban I. Gavrila
VDG Inc.
6009 Brookside Drive
Chevy Chase, MD. 20815

David Ferraiolo
NIST
U.S. Dept. of Commerce
Gaithersburg, MD. 20798

Abstract

In this paper we define formally a wide variety of separation-of-duty (SoD) properties, which include the best known to date, and establish their relationships within a formal model of role-based access control (RBAC). The formalism helps remove all ambiguities of informal definition, and offers a wide choice of implementation strategies. We also explore the composability of SoD properties and policies under a simple criterion. We conclude that practical implementation for SoD policies requires new methods and tools for security administration even within applications that already support RBAC, such as most database management systems.

1 Introduction

As a security principle, separation of duty (SoD) has had wide application in business, industry, and government [3, 4, 7]. Its purpose is to ensure that failures of omission or commission within an organization are caused only by collusion among individuals and, therefore, are riskier and less likely, and that chances of collusion are minimized by assigning individuals of different skills or divergent interests to separate tasks. For example, SoD is enacted whenever conflict of interest may otherwise arise in assignment of tasks within an organization.

As an application-design principle, SoD requires the following three well-understood design and implementation steps:

Integrity Property Definition. Within an application domain, the aim of SoD is defined by integrity properties. These properties may require that each application include independent, redundant functions whose results must match to enable a sensitive task (e.g., the results of two orthogonal single-entry accounting activities must match; two persons must approve the execution of an action such as signing a check, installing a crypto key, or launching a missile). Or, they may require that an application enact conflict-of-interest resolution by establishing balances and checks among its tasks (e.g., different users are required to perform asset accounting and control, or system security administration and auditing).

Application Design. The objects and operations of an application subject to SoD are partitioned to implement, maintain, and verify the integrity properties. For example, application partitioning may separate the operations that perform accounts-payable, purchasing, and payroll tasks. It may further separate check reading/writing from signing operations within both accounts-payable and payroll tasks, and purchase-order reading/writing from signing operations within purchasing tasks.

User Assignment to Application Partitions. Users of different skills or interests are assigned to operate in different application partitions. These assignments may last for limited periods of time, and may change dynamically. By performing such assignments, application administration ensures that user collusion is required to breach integrity properties, and that chances of collusion are minimized.

Despite its importance as a security principle and its well-understood application in business, industry, and government, few computer systems have supported SoD as a security policy to date [10, 15]. We attribute the lack of wide-spread support to three separate reasons. First, SoD is an inherently *application-oriented* policy and, thus, has been perceived to yield limited payoff for operating systems and networks, since it cannot be used as a global, system-wide security policy. Second, when the SoD principle is interpreted within different applications, it may yield many different SoD policies and, thus, support of all policies is perceived to require both substantial system flexibility [15] and recurrent administrative costs -- an unmistakable recipe for both system-vendor and market resistance in the absence of advanced policy-administration tools. Third, most SoD policies proposed to date have been only informally

defined and, therefore, subject to ambiguous or incomplete specifications, and limited assurance. As a consequence, both relationships among SoD properties and policy composability -- an important requirement for all application-oriented policies -- could not be easily established.

The perceived disadvantages of SoD as a family of application-oriented policies can be mitigated by two factors. First, growing interest in internet applications and their security may lead to the development of versatile SoD policies and administrative tools at a cost made reasonable by the economies of scale. Second, better understanding of SoD policies for computer systems via precise, possibly formal, definition may remove the aura of complexity associated with these policies, thereby facilitating their acceptance.

In this paper we define a set of SoD policies (Sections 2 and 4), which includes the best-known ones [15], and establish their relationships within a formal model of role-based access control (RBAC) [8]. The formalism (Sections 3 and 4) helps remove ambiguities of informal definition and offers a wide choice of implementation strategies. We also explore the composability of these policies under a simple criterion, and identify classes of policies that are, or are not, composable (Section 5). We conclude (Section 6) that practical implementation for SoD policies requires new methods and tools for security administration, even within applications that support RBAC, such as most database management systems [12].

2 Specification of SoD Policies

In this section, we define the structure of a SoD policy as a conjunction of constituent properties (or predicates) of system states and state transitions, or of command sequences (or traces) executed by systems. To view security policies as conjunctions of properties is appealing because it helps determine security-policy effectiveness [11] -- by assigning each property a role in countering a threat -- and relative strength -- by (partially) ordering the threats countered by policy properties. However, we adopt this view because we are interested in defining security policies by property composition. This allows us to use similar composition criteria for incremental property addition to, removal from, or modification of extant policies. For example, we define SoD policies both by incremental addition of new properties to conjunctions of RBAC properties and by composing extant SoD policies.

2.1 Dependencies Among Policy Properties

Although viewing security policies as compositions of security properties (i.e., by conjunction) is appealing, care must be exercised because the composition of *independent* security properties does not necessarily define a policy. The reason for this is that the properties of a policy are *not* independent; i.e., both individual and groups of properties may depend on other properties of the conjunction to counter a specific set of threats. Ignoring dependencies among policy properties has undesirable effects[1]. For example, the inadvertent omission or incorrect modification of any property that is depended upon by others may cause the policy to become ineffective, undefined or incorrectly defined, or empty. Furthermore, analysis of policy effectiveness, relative strength, and composability may yield inconsistent or incorrect results without analysis of property dependencies. In Section 5, we show that property omission can cause policies which are otherwise not composable to appear to be composable. (For similar reasons, the identification of dependencies among the properties of cryptographic protocols is also considered important [11].)

To illustrate the notion of dependency among policy properties, consider the following three types of properties: *access-attribute* (AT), *access-authorization* (AA), *and access-management* (AM) properties. Access attributes include subject and object attributes (e.g., user, group, role, location identifiers, secrecy and integrity levels, time-of-access intervals), and AT properties typically establish invariant relationships among attributes (e.g., lattices of secrecy and integrity levels, user-group membership invariants, inheritance of role permissions). AA properties determine whether a subject's current access attributes satisfy the conditions for accessing an object given the current object attributes. In contrast, AM properties include conditions under which subjects are granted or revoked attributes for accessing objects, subjects and objects are created or destroyed, objects are encapsulated within protected subsystems. In general, AM properties characterize commands and constraints that bring the system to desired states, whereas AA properties characterize constraints that ensure that the system remains in desired states.

[1] The notion of dependency among the policy properties (a.k.a. dependencies among policy "functional" requirements) is neither new nor novel. It has already received extensive coverage in the security-standards literature [5, 6, 17] where templates for defining policy properties are provided to enable policy analysis and evaluation.

A typical dependency arises between AA and AM properties because the values of the attributes used by the former for access decisions are determined by the latter (i.e., AA properties have a "uses" dependency on AM properties). Hence, whether an AA property holds in a state depends not only on the property (predicate) itself but also on the defined AM properties. Both AA and AM properties have "uses" dependencies on AT properties. Furthermore, AM properties also have "uses" dependencies on AA properties whenever object attributes themselves are treated as objects and, hence, become the target of access authorization. Techniques to redefine policy properties to avoid this cyclic dependency, and other undesirable ones, are presented elsewhere [6].

Groups of policy properties may also depend upon individual properties used in the same conjunction. For example, the property $P = AT \wedge AA \wedge AM$, depends on an AM property, denoted by $Admin(P)$, which requires that administrative commands have the ability to bring the system from an arbitrary state to a state (or any state) that satisfies P. $Admin(P)$ is formally defined in Section 5. Unless $Admin(P)$ is satisfied, the policy defined by property P is *empty* because the system operating under P can neither start from, or recover to, a secure state [16] nor reach a secure state from another secure state under administrative control.

For *application-oriented* policies, such as SoD, the composition of independent security properties does not necessarily define a *usable* policy. To be useful, property $P = AT \wedge AA \wedge AM$ must be compatible with the application for which it was designed. Informally, this means that P may not unjustifiably deny the execution of the application; e.g., an AM property should not grant users too few, or inconsistent, permissions for application execution. This property, denoted by $Compat(P, App)$, where App is the application, is formally defined in Section 5.

2.2 Structure of SoD Policies in RBAC Systems

A security policy $\mathcal{P}$ can be defined as:

$$\mathcal{P} = P \wedge Admin(P),$$

where $P = AT \wedge AA \wedge AM$. The property P itself may have other properties in addition to $Admin(P)$; e.g., application-oriented policies, such as SoD, also include property $Compat(P, App)$. We specify SoD policies as incremental conjunctions of properties to RBAC policies. That is,

$$\mathcal{SoD\text{-}P} = SoD\text{-}P \wedge Admin(SoD\text{-}P) \wedge Compat(SoD\text{-}P, App) \wedge \mathcal{RBAC\text{-}P}$$

where $\mathcal{RBAC\text{-}P} = RBAC\text{-}P \wedge Admin(RBAC\text{-}P)$, and both *SoD-P* and *RBAC-P* are conjunctions of AT, AM, and AA properties. Some, but not all, properties of *SoD-P* may be empty.

The specification of SoD policies as incremental additions of specific properties to RBAC policies is both useful and justified in practice. It is useful because a large number of properties that are common to all SoD policies are part of RBAC models. For example, RBAC formal models [8] include both flexible AA and AM properties and, hence, AM and AA properties of SoD can be defined as incremental properties (i.e., predicates on the types, functions, states, and state transitions) of RBAC. In practice, most SoD policies are implemented atop systems that support RBAC policies. RBAC enables the partitioning of application operations and objects, and also the selective assignment of roles to application operations and users (two of the key steps in SoD policy design presented in Section 1).

3 Secure RBAC Systems and Applications

In this section we define the types, functions, and properties of a RBAC system that are necessary to define SoD properties. The formal RBAC model, which consists of a conjunction of RBAC policy properties, is presented elsewhere [8].

We consider a RBAC system to be defined by a state machine model. We denote the set of system states by *STATES*, the set of subjects by *SUBJECTS*, the set of users by *USERS*, the set of operations by *OPERATIONS*, and the set of objects by *OBJECTS*. A RBAC system is characterized by the fact that a user's rights to access objects are defined by the user's membership to a "role" and by the roles' permissions to perform operations on those objects. Hence, a *role* is a collection of operations on object sets. The class of roles, *ROLES*, is a subset of $2^{OPERATIONS \times 2^{OBJECTS}}$. The function

$$auth : STATES \times ROLES \times OBJECTS \rightarrow 2^{OPERATIONS}$$

defines the operations allowed to each role in each state of the system:

$$\forall s \in STATES, \forall op \in OPERATIONS, \forall r \in ROLES,$$
$$\forall obj \in OBJECTS, op \in auth(s, r, obj) \Leftrightarrow$$
$$\exists objset \subseteq OBJECTS : obj \in objset \wedge (op, objset) \in r.$$

The function $role\_members : ROLES \rightarrow 2^{USERS}$ defines the users *assigned* to a given role.

The function $subject\_user : SUBJECTS \rightarrow USERS$ returns the user associated with the subject.

The function

$$subject\_roles : STATES \times SUBJECTS \rightarrow 2^{ROLES}$$

returns the roles assumed by a user in a given state while executing a given subject. These roles must have been assigned to the subject's user.

The function

$current\_role\_set: STATES \times USERS \rightarrow 2^{ROLES}$

is defined as follows:

$$\forall s \in STATES, \forall u \in USERS,$$
$$current\_role\_set(s, u) = \bigcup_{\substack{S \in SUBJECT \\ subject\_user(S)=u}} subject\_roles(S).$$

If $r \in current\_role\_set(s, u)$, then we say that the role r is *enabled* or *active* for the user u in state s.

Each *state transition* is defined by a command of the form $op(s_1, S, obj, s_2)$, where the subject S performs the operation op on the object or objects denoted by obj, thereby changing the system state from s_1 to s_2. Access authorization in RBAC requires that the state transition $op(s_1, S, obj, s_2)$ take place only if at least one of the roles in $subject\_roles(s_1, S)$ has the permission to perform the operation op on the object, or objects, obj. Formally,

$op(s_1, S, obj, s_2) \Rightarrow$

$\exists r \in subject\_roles(s_1, S): op \in auth(s_1, r, obj)$.

A *command sequence* is $op_1(s_0, S_1, obj, s_1) \cdot op_2(s_1, S_2, obj_2, s_2) \cdot \ldots$, where "$\cdot$" is the concatenation operator, and s_0 is the *start state*. We denote the set of start states of a command sequence by $STATES_0$. A finite command sequence σ may be extended to an infinite one by adding "no-op" commands, that do nothing and preserve the system state. $\hat{\sigma}$ denotes the extended command sequence. If s_0 is a start state, $\hat{s}_0$ denotes the command sequence starting in s_0 and consisting only of no-op commands. We denote the set of command sequences of a system with the start states in $STATES_0$ by Ω_0. Whenever $STATES_0 = STATES$, we drop the subscript "$_0$".

A *tranquil command* is a command that does not alter the security attributes/data of the system (e.g., creation, deletion, or update of roles). A *tranquil command sequence* is a command sequence consisting only of tranquil commands. We denote the set of tranquil command sequences of a system with the start states in $STATES_0$ by Σ_0. Whenever $STATES_0 = STATES$, we drop the subscript "$_0$".

A *secure state* is a state that satisfies some "state" properties. A *secure command* is a command that satisfies some "transition" properties. A *reachable state* is a state appearing in a command sequence. A system is *secure* if all its reachable states are secure and all commands used in its command sequences are secure. In this paper, we consider only secure RBAC systems.

The partial function:

$access\_history: STATES \times STATES \times USERS \times 2^{ROLES} \times OBJECTS \rightarrow 2^{OPERATIONS}$

returns the operations performed by RBAC users in all states between s_0 and s, where state s is reachable from state s_0. It has the following properties:

1. $\forall s \in STATES, \forall u \in USERS, \forall r \in ROLES,$
$\forall obj \in OBJECTS, access\_history(s, s, u, \{r\}, obj) = \varnothing.$

2. $\forall s_0, s \in STATES, \forall u \in USERS, \forall roleset \subseteq ROLES,$
$\forall obj \in OBJECTS, access\_history(s_0, s, u, roleset, obj) =$
$\bigcup_{r \in roleset} access\_history(s_0, s, u, \{r\}, obj).$

3. $\forall s_0, s_1, s_2 \in STATES, \forall op \in OPERATIONS,$
$\forall S \in SUBJECTS, \forall u \in USERS, \forall r \in ROLES,$
$op(s_1, S, obj, s_2) \wedge subject\_user(S) = u \wedge$
$r \in subject\_roles(S) \Rightarrow$
$access\_history(s_0, s_2, u, \{r\}, obj) =$
$access\_history(s_0, s_1, u, \{r\}, obj) \cup \{op\}.$

An *application* is a tuple $App = [ObjSet, OpSet, Plan]$, where $ObjSet \subseteq OBJECTS$ and $OpSet \subseteq OPERATIONS$. $Plan$ is the *execution plan* of the application and consists of a finite set of pairs $\{(obj_i, op_i) | i \in \{1,\ldots,n\}\}$, where n is a natural number, obj_i is one or more objects of $ObjSet$, and $op_i \in OpSet$.

Given two applications $App_i = [ObjSet_i, OpSet_i, Plan_i]$, i=1, 2, we denote the new application $[ObjSet_1 \cup ObjSet_2, OpSet_1 \cup OpSet_2, Plan_1 \cup Plan_2]$ by $App_1 \sqcup App_2$.

A command sequence $\sigma \in \Sigma_0$ *executes* the application App (App is executed by σ, or σ is an execution of App), if for any pair (obj, op) in the App's execution plan there is a command $op(s_k, S, obj, s_{k+1})$ in σ. For simplicity, we omit other types of application executions and execution plans; e.g., executions and plans that include order, or that exclude redundant operations and privileges to objects to help satisfy the "least privilege" principle. We also use App to denote the set of all executions σ of App.

A user needs permissions to execute the operations of an application App, and since these permissions as given by roles, the user must assume the necessary roles to execute App. The AM properties of SoD policies specify the assignments of roles to application operations and users.

4 SoD Properties in Secure RBAC Systems

In this section, we define a variety of SoD properties, their relationships in secure RBAC systems, and their composition based on property conjunctions. As usual [15], we distinguish between "static" and "dynamic" SoD properties.

Clark and Wilson defined the static SoD by the rule that: "each user must be permitted to use only certain ... transactions" [4]. In a RBAC environment, their definition is expressed as follows:

Static Separation of Duty (*SSoD*). Let App be an application and $RoleSet$ its assigned roles in a secure

RBAC system. $\sigma \in \Sigma_0$ satisfies the *SSoD* property with respect to *App* if any two distinct roles in *RoleSet* do not have common members. Such roles are said to be *restricted*. Formally,

$\sigma \in SSoD(RoleSet, App) \Leftrightarrow$
$\forall r_1, r_2 \in RoleSet, r_1 \neq r_2 \Rightarrow$
$role\_members(r_1) \cap role\_members(r_2) = \emptyset$.

We obtain a stronger version of this property by adding the requirement that the target object sets of two restricted roles be *disjoint*:

Strict Static Separation of Duty (*SSSoD*). Let *App*=[*ObjSet*, *OpSet*, *Plan*] be an application and *RoleSet* its assigned roles in a secure RBAC system. $\sigma \in \Sigma_0$ satisfies the *SSSoD* property with respect to *App* if any two distinct roles in *RoleSet*: a) do no have common members, and b) are not authorized to perform operations in *OpSet* on the same object of the application. Formally:

$\sigma \in SSSoD(RoleSet, App) \Leftrightarrow$
($\forall s$ state of σ, $\forall r_1, r_2 \in RoleSet, r_1 \neq r_2 \Rightarrow$
$role\_members(r_1) \cap role\_members(r_2) = \emptyset \wedge$
$\{o \in ObjSet \mid auth(s, r_1, o) \cap OpSet \neq \emptyset\} \cap$
$\{o \in ObjSet \mid auth(s, r_2, o) \cap OpSet \neq \emptyset\} = \emptyset$).

We obtain a still stronger version by adding the requirement that each role execute *only one step* (operation) of the application [14]:

1-step Strict Static Separation of Duty (1sS*SSoD*). Let *App*=[*ObjSet*, *OpSet*, *Plan*] be an application and *RoleSet* its assigned roles in a secure RBAC system. $\sigma \in \Sigma_0$ satisfies the *1sSSSoD* property with respect to *App* if a) any two distinct roles in *RoleSet* do no have common members; b) any role in *RoleSet* is authorized to perform at most one operation of *OpSet* on application's objects; and c) any two distinct roles in *RoleSet* are not authorized to perform operations in *OpSet* on the same object of the application. Formally:

$\sigma \in 1sSSSoD(RoleSet, App) \Leftrightarrow$
($\forall s$ state of σ, $\forall r, r_1, r_2 \in RoleSet$, $\forall obj \in ObjSet$,
$|auth(s, r, obj) \cap OpSet| \leq 1 \wedge$
($r_1 \neq r_2 \Rightarrow$
$role\_members(r_1) \cap role\_members(r_2) = \emptyset \wedge$
$\{o \in ObjSet \mid auth(s, r_1, o) \cap OpSet \neq \emptyset\} \cap$
$\{o \in ObjSet \mid auth(s, r_2, o) \cap OpSet \neq \emptyset\} = \emptyset$).

Clark and Wilson [4], and then others, defined several "dynamic separation of duty" properties. In an RBAC system, a dynamic SoD property with respect to the roles assumed by the users ("active" roles) can be defined as follows [7]:

Dynamic Separation of Duty (DSoD). Let *App*=[*ObjSet*, *OpSet*, *Plan*] be an application and *RoleSet* its assigned roles in a secure RBAC system. $\sigma \in \Sigma_0$ satisfies the *DSoD* property with respect to *App* if, in any state of σ, there is no user with two distinct roles in RoleSet enabled (active). Formally,

$\sigma \in DSoD(RoleSet, App) \Leftrightarrow$
($\forall s$ state of σ, $\forall r_1, r_2 \in RoleSet$, $\forall u \in USERS$,
$r_1 \neq r_2 \wedge r_1 \in current\_role\_set(s, u) \Rightarrow$
$r_2 \notin current\_role\_set(s, u)$).

Nash and Poland [13] introduced the object-based, dynamic SoD, as a more flexible and realistic alternative to the static SoD. However, their informal definition [13, 15] does not specify precisely which objects, operations, roles are subjected to the object-based SoD condition. Our definition removes these ambiguities.

Object-based Dynamic Separation of Duty (*ObjDSoD*). Let *App*=[*ObjSet*, *OpSet*, *Plan*] be an application and *RoleSet* its assigned roles in a secure RBAC system. $\sigma \in \Sigma_0$ satisfies the *ObjDSoD* property with respect to *App* if, any user which performs an operation in *OpSet* on an object of *ObjSet* in a role of *RoleSet* as a command of σ has not already performed another operation of *OpSet* on the same object in a role of *RoleSet*. Formally,

$\sigma \in ObjDSoD(RoleSet, App) \Leftrightarrow$
($\forall s_0, s_1, s_2 \in STATES$, $\forall r \in RoleSet$, $\forall op \in OpSet$,
$\forall obj \in ObjSet$, $\forall S \in SUBJECTS$,
σ starts in $s_0 \wedge$
$op(s_1, S, obj, s_2)$ is in $\sigma \wedge$
$subject\_roles(s_1, S) \cap RoleSet \neq \emptyset \Rightarrow$
$access\_history(s_0, s_1, subject\_user(S), \{r\}, obj) \cap$
$OpSet \subseteq \{op\}$).

The following two properties are static variants of the object-based dynamic SoD, the first for set of roles with common members, and the second for single roles.

Object-based Static Separation of Duty (ObjSSoD). Let *App*=[*ObjSet*, *OpSet*, *Plan*] be an application and *RoleSet* its assigned roles in a secure RBAC system. $\sigma \in \Sigma_0$ satisfies the *ObjSSoD* property with respect to *App* if, in any state σ, no group of application roles with a common assigned user is authorized to perform more than one operation on each object of the application. Formally,

$\sigma \in ObjSSoD(RoleSet, App) \Leftrightarrow$
$\forall s$ state of σ, $\forall RoleSubset \subseteq RoleSet$, $\forall op_1, op_2 \in OpSet$,
$\forall obj \in ObjSet$,
$op_1 \neq op_2 \wedge \bigcap_{r \in RoleSubset} role\_members(r) \neq \emptyset \Rightarrow$
$\{op_1, op_2\} \not\subseteq \bigcup_{r \in RoleSubset} auth(s, r, obj)$.

Per-Role Object-based Static Separation of Duty (*RObjSSoD*). Let *App*=[*ObjSet*, *OpSet*, *Plan*] be an application and *RoleSet* its assigned roles in a secure RBAC system. $\sigma \in \Sigma_0$ satisfies the *RObjSSoD* property with respect to *App* if, in any state of σ, no role in *RoleSet*

is authorized to perform more than one operation of *OpSet*. Formally,

$\sigma \in ROpSSoD(RoleSet, App) \Leftrightarrow$

$(\forall s$ state of σ, $\forall r \in RoleSet$, $\forall op_1, op_2 \in OpSet$,

$\forall obj \in ObjSet$,

$op_1 \neq op_2 \Rightarrow \{op_1, op_2\} \not\subseteq auth(s, r, obj))$.

Another flexible alternative to static SoD is the operational SoD. The following property is (a corrected version of that) presented by Ferraiolo, Cugini and Kuhn [7]:

Operational Static Separation of Duty (*OpSSoD*). Let *App*=[*ObjSet*, *OpSet*, *Plan*] be an application and *RoleSet* its assigned roles in a secure RBAC system. $\sigma \in \Sigma_0$ satisfies the *OpSSoD* property with respect to *App* if, in any state of σ, any subset of roles in *RoleSet* with a common member is not authorized to perform all the operations of *OpSet* (if more than one), regardless of the target object. Formally,

$\sigma \in OpSSoD(RoleSet, App) \Leftrightarrow$

$\forall s$ state of σ, $\forall RoleSubset \subseteq RoleSet$,

$|OpSet| \geq 2 \wedge \bigcap_{r \in RoleSubset} role\_members(r) \neq \varnothing \Rightarrow$

$OpSet \not\subseteq \bigcup_{\substack{r \in RoleSubset \\ obj \in ObjSet}} auth(s, r, obj)$.

The following variant of operational separation of duty is obtained by applying the operational separation of duty to single roles.

Per-Role Operational Static Separation of Duty (*ROpSSoD*). Let *App*=[*ObjSet*, *OpSet*, *Plan*] be an application and *RoleSet* its assigned roles in a secure RBAC system. $\sigma \in \Sigma_0$ satisfies the *OpSSoD* property with respect to *App* if, in any state of σ, no application role is authorized to perform all the operations of the application regardless of the target object. Formally,

$\sigma \in ROpSSoD(RoleSet, App) \Leftrightarrow$

$\forall s$ state of σ, $\forall r \in RoleSet$,

$|OpSet| \geq 2 \Rightarrow OpSet \not\subseteq \bigcup_{obj \in ObjSet} auth(s, r, obj)$.

Operation separation of duty has the following dynamic variant.

Operational Dynamic Separation of Duty (*OpDSoD*). Let *App*=[*ObjSet*, *OpSet*, *Plan*] be an application and *RoleSet* its assigned roles in a secure RBAC system. $\sigma \in \Sigma_0$ satisfies the *OpDSoD* property with respect to *App* if, in any state of σ, any subset of roles in *RoleSet* enabled for the same user is not authorized to perform all the operations of *OpSet* (if more than one), regardless of the target object. Formally,

$\sigma \in OpDSoD(RoleSet, App) \Leftrightarrow$

$\forall s$ state of σ, $\forall u \in USERS$, $\forall RoleSubset \subseteq RoleSet$,

$|OpSet| \geq 2 \wedge RoleSubset \subseteq current\_role\_set(s, u) \Rightarrow$

$OpSet \not\subseteq \bigcup_{\substack{r \in RoleSubset \\ obj \in ObjSet}} auth(s, r, obj)$.

Simon and Zurko [15] have generalized both operational and object-based dynamic separation of duty as the history-based separation of duty. The formal version of their property follows.

History-based Dynamic Separation of Duty (*HDSoD*). Let *App*=[*ObjSet*, *OpSet*, *Plan*] be an application and *RoleSet* its assigned roles in a secure RBAC system. $\sigma \in \Sigma_0$ satisfies the *HDSoD* property with respect to *App* if in σ, the same user cannot perform all the operations in *OpSet* on the same object of *ObjSet* in roles assigned to the application. Formally,

$\sigma \in HDSoD(RoleSet, App) \Leftrightarrow$

$(\forall s_0, s_1, s_2 \in STATES$, $\forall op \in OpSet$, $\forall obj \in ObjSet$,

$\forall S \in SUBJECTS$, $\forall u \in USERS$

σ starts in $s_0 \wedge op(s_1, S, obj, s_2)$ is in $\sigma \wedge$

$u = subject\_user(S) \wedge$

$subject\_roles(s_1, S) \cap RoleSet \neq \varnothing \wedge |OpSet| \geq 2 \Rightarrow$

$OpSet \not\subseteq access\_history(s_0, s_1, u, RoleSet, obj) \cup \{op\})$.

The following theorem shows that, for all SoD properties except the Object-based Dynamic SoD and History-based Dynamic SoD, if the start state of a command sequence satisfies the state invariants of the SoD property, then the command sequence satisfies the SoD property. (The proofs of all theorems of this paper are included in the Appendix.)

Theorem 1. Let *App* be an application in a secure RBAC system, and *P* one of the following SoD-P properties based on *App*: *1sSSSoD*, *SSSoD*, *SSoD, DSoD*, *RObjSSoD*, *ObjSSoD*, *ROpSSoD*, *OpSSoD*, *OpDSoD*. Let $s_0 \in STATES_0$, and $\sigma \in \Sigma_0$ starting in s_0. If $\hat{s}_0 \in P$ then $\sigma \in P$. As a corollary, if $\hat{s}_0 \in P$ for all $s_0 \in STATES_0$, then $P = \Sigma_0$.

The following two theorems express the Object- and History-based Dynamic SoD properties by equivalent, more intuitive properties:

Theorem 2. Let *App*=[*ObjSet*, *OpSet*] be an application in a secure RBAC system, RoleSet its assigned roles, and $\sigma \in \Sigma_0$. $\sigma \in ObjDSoD(RoleSet, App)$ if and only if all distinct operations in *OpSet* performed in states of σ on the same object of *ObjSet* in roles of *RoleSet* were performed by distinct users.

Theorem 3. Let *App*=[*ObjSet*, *OpSet*] be an application in a secure RBAC system, *RoleSet* its assigned roles, and $\sigma \in \Sigma_0$. The following statements are equivalent:

1. $\sigma \in HDSoD(RoleSet, App)$;
2. if σ executes *App*, then the operations of *App* (i.e., the operations in *OpSet* on objects in *ObjSet* performed in roles in *RoleSet*) are executed by at least two distinct users.

The definition of the wide variety of SoD properties above raises the question as to which policy to implement in a secure RBAC system, and which of the implemented policies to enforce in a given application. Although these questions cannot be answered independently of the application and system context, it is helpful to examine the relationships among these properties and property composability independent of their context of use. These relationships help in making the choice between implementing a stronger property, or a set of composable properties, instead of a more flexible property that may be harder to implement or administer. For example, one may choose to implement either Object-based Static SoD or the composition of per-Role Object-based Static SoD and Dynamic SoD, instead of Object-based Dynamic SoD (viz., Figure 1).

The following two theorems summarize the relationships among, and composability of, the SoD properties defined in this section.

Theorem 4. Let *App* be an application and *RoleSet* its assigned roles in a secure RBAC system. Assume that, in each start state, the set *role_members*(r) of each role $r \in RoleSet$ is not empty. Then the following properties hold:

1. *1sSSSoD*(*RoleSet*, *App*) ⇒ *SSSoD*(*RoleSet*, *App*).
2. *1sSSSoD*(*RoleSet*, *App*) ⇒ *ObjSSoD*(*RoleSet*, *App*).
3. *1sSSSoD*(*RoleSet*, *App*) ⇒ *OpSSoD*(*RoleSet*, *App*).
4. *SSSoD*(*RoleSet*, *App*) ⇒ *SSoD*(*RoleSet*, *App*).
5. *SSoD*(*RoleSet*, *App*) ⇒ *DSoD*(*RoleSet*, *App*).
6. *ObjSSoD*(*RoleSet*, *App*) ⇒ *RObjSSoD*(*RoleSet*, *App*);
7. *SSoD*(*RoleSet*, *App*) ⇒
   (*ObjSSoD*(*RoleSet*, *App*) ⇔
   RObjSSoD(*RoleSet*, *App*)).
8. *RObjSSoD*(*RoleSet*, *App*) ∧ *DSoD*(*RoleSet*, *App*) ⇒
   ObjDSoD(*RoleSet*, *App*).
9. *ObjSSoD*(*RoleSet*, *App*) ⇒ *ObjDSoD*(*RoleSet*, *App*).
10. *OpSSoD*(*RoleSet*, *App*) ⇒ *ROpSSoD*(*RoleSet*, *App*);
11. *SSoD*(*RoleSet*, *App*) ⇒
    (*OpSSoD*(*RoleSet*, *App*) ⇔
    ROpSSoD(*RoleSet*, *App*)).
12. *ROpSSoD*(*RoleSet*, *App*) ∧ *DSoD*(*RoleSet*, *App*) ⇒
    OpDSoD(*RoleSet*, *App*).
13. *OpSSoD*(*RoleSet*, *App*) ⇒ *OpDSoD*(*RoleSet*, *App*).
14. *ObjDSoD*(*RoleSet*, *App*) ⇒ *HDSoD*(*RoleSet*, *App*).
15. *OpDSoD*(*RoleSet*, *App*) ⇒ *HDSoD*(*RoleSet*, *App*).

Figure 1 illustrates the relationships among SoD properties. To define property composition, let App_1, App_2 be two applications in a secure RBAC system, and $RoleSet_1$, $RoleSet_2$ their assigned roles, and let $P_1 = P_1(RoleSet_1, App_1)$ and $P_2 = P_2(RoleSet_2, App_2)$ be any two of the SoD properties defined in this section.

Definition. We say that properties P_1 and P_2 are composable if $P_1 \cap P_2 \neq \varnothing$ whenever $P_1 \neq \varnothing$ and $P_2 \neq \varnothing$.

If P_1 and P_2 are composable, then $P_1 \circ P_2$ denotes their composition. As a predicate, $P_1 \circ P_2 = P_1 \wedge P_2$, and as a set of command sequences, $P_1 \circ P_2 = P_1 \cap P_2$.

Theorem 5. In a secure RBAC system that has:
(1) arbitrary start states, $STATES_0 = STATES$, any two SoD properties are composable;
(2) constrained start states, $STATES_0 \subsetneqq STATES$, the SoD properties are not necessarily composable.
(3) a single start state, $|STATES_0| = 1$, any two SoD properties are composable.

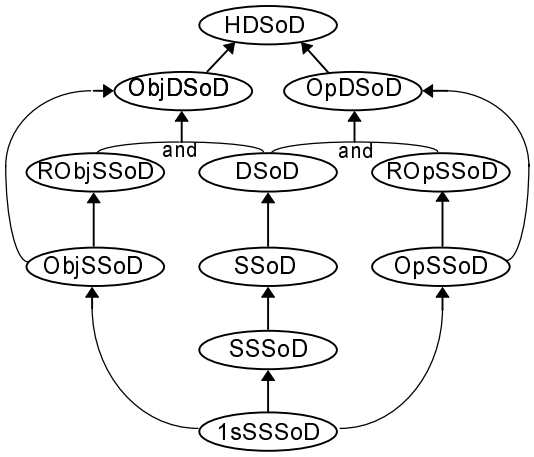

Figure 1. Relationships among SoD Properties

The composition of SoD properties is, actually, a conjunction of predicates, and an intersection of sets. Hence, the following relations hold for any set of start states and any SoD properties P, Q, R:

Idempotency. P is composable with itself, and $P \circ P = P$.
Monotonicity. If P is composable with Q, then $P \circ Q \Rightarrow P$.
Commutativity. If P is composable with Q, then Q is composable with P and $P \circ Q = Q \circ P$.
Strengthening. If $P \Rightarrow Q$, P is composable with R, and Q is composable with R, then $P \circ R \Rightarrow Q \circ R$. Note that the composability of P with R does not lead to the composability of Q with R (unless $P \neq \varnothing$).
Associativity. If P is composable with Q, $P \circ Q$ is composable with R, Q is composable with R, and P is composable with $Q \circ R$, then $(P \circ Q) \circ R = P \circ (Q \circ R)$. Note that the composability of P with Q and of $P \circ Q$ with R does not lead to the composability of Q with R (unless $P \neq \varnothing$).

5 A Simple Composition Criterion

In this section, we define a simple composition criterion for SoD policies in RBAC systems and illustrate the use of the criterion with two examples.

In Section 2.2, we defined an SoD policy as follows:

$\mathcal{SoD\text{-}P}$ = *SoD-P*∧*Admin*(*SoD-P*)∧*Compat*(*SoD-P*, *App*) ∧ $\mathcal{RBAC\text{-}P}$

where $\mathcal{RBAC\text{-}P}$ = *RBAC-P* ∧ *Admin*(*RBAC-P*), and both *SoD-P* and *RBAC-P* are conjunctions of *AT, AM*, and *AA* properties. Since we assumed that all SoD policies are implemented in secure RBAC systems, $\mathcal{RBAC\text{-}P}$ is satisfied and, hence,

$\mathcal{SoD\text{-}P}$ = *SoD-P*∧*Admin*(*SoD-P*)∧*Compat*(*SoD-P*, *App*), where *Admin*(*SoD-P*) and *Compat*(*SoD-P*, *App*) are defined as follows:

Definition. *Compat*(*SoD-P, App*) is satisfied if and only if *SoD-P* ∩ *App* ≠∅; i.e., *SoD-P* includes at least a command sequence that executes the application *App*.

Example 2 below shows that, unless *Compat*(*P*, *App*) is satisfied, composition of two policies cannot guarantee application executability. Hence, omitting *Compat*(*P*, *App*) can cause policies which otherwise are not composable to appear to be composable.

Definition. *Admin*(*SoD-P*) is satisfied if and only if ∀*s*∈*STATES*, ∃s_0∈*STATES*$_0$, ∃ω∈Ω such that ω starts in *s* ∧ ω reaches s_0 ∧ $\hat{s}_0$ ∈ *SoD-P*; i.e., starting in an arbitrary state, the administrative commands have the ability to bring the system in a state that satisfies property *SoD-P*[2].

A variant of *Admin*(*SoD-P*) might require that *any* state s_0∈*STATES*$_0$ such that $\hat{s}_0$ ∈ *SoD-P* be reachable from any other state.

Note that the predicate "ω reaches s_0" of *Admin*(*SoD-P)* is not trivially satisfied; e.g., the system may not provide all the administrative commands to ensure that certain states of *STATES*$_0$ can be reached.

As a set, $\mathcal{SoD\text{-}P}$ = *SoD-P* whenever *Admin*(*SoD-P*) and *Compat*(*SoD-P*, *App*) are satisfied, and ∅ otherwise. Whether *Admin*(*SoD-P*) and *Compat*(*SoD-P*, *App*) are satisfied requires the analysis of both command and application code.

Let App_1 and App_2 be two applications of a secure system, and let $\mathcal{P}_1$ = P_1∧*Admin*(P_1)∧Compat(P_1, App_1), $\mathcal{P}_2$ = P_2∧*Admin*(P_2)∧Compat(P_2, App_2) be two SoD policies for applications App_1 and App_2.

Definition. Let $\mathcal{P}_1$ °$\mathcal{P}_2$ be the SoD policy (P_1∧P_2)∧ *Admin*(P_1∧P_2)∧Compat(P_1∧P_2, App_1⊔App_2). We say that $\mathcal{P}_1$ is *composable* with $\mathcal{P}_2$ if and only if $\mathcal{P}_1$°$\mathcal{P}_2$ ≠ ∅ whenever $\mathcal{P}_1$≠∅ and $\mathcal{P}_2$≠∅. Hence,

$\mathcal{P}_1$ is *composable* with $\mathcal{P}_2$ ⇔
(∃σ_1∈P_1: σ_1 executes App_1 ∧ *Admin*(P_1) ∧
∃σ_2∈P_2: σ_2 executes App_2 ∧ Admin(P_2) ⇒
∃σ∈P_1∩P_2: σ executes App_1, App_2 ∧ Admin(P_1∧P_2)).

If $\mathcal{P}_1$ is composable with $\mathcal{P}_2$, then $\mathcal{P}_1$°$\mathcal{P}_2$ is called the *composition* of $\mathcal{P}_1$ with $\mathcal{P}_2$ or the *emerging policy*, and the set command sequences is $\mathcal{P}_1$∩$\mathcal{P}_2$.

The following properties of composition hold:

Idempotency: any policy $\mathcal{P}$ is composable with itself, and $\mathcal{P}$°$\mathcal{P}$ = $\mathcal{P}$.

Commutativity: for any two policies, $\mathcal{P}_1$ and $\mathcal{P}_2$, if $\mathcal{P}_1$ is composable with $\mathcal{P}_2$, then $\mathcal{P}_2$ is composable with $\mathcal{P}_1$ and $\mathcal{P}_1$°$\mathcal{P}_2$ = $\mathcal{P}_2$°$\mathcal{P}_1$.

Monotonicity: for any two policies $\mathcal{P}_1$ and $\mathcal{P}_2$, if $\mathcal{P}_1$ is composable with $\mathcal{P}_2$, then $\mathcal{P}_1$°$\mathcal{P}_2$ ⇒ $\mathcal{P}_1$.

Associativity: for any three policies, $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, if $\mathcal{P}_1$ is composable with $\mathcal{P}_2$, $\mathcal{P}_1$°$\mathcal{P}_2$ is composable with $\mathcal{P}_3$, $\mathcal{P}_2$ is composable with $\mathcal{P}_3$, and $\mathcal{P}_1$ is composable with $\mathcal{P}_2$°$\mathcal{P}_3$, then ($\mathcal{P}_1$°$\mathcal{P}_2$)°$\mathcal{P}_3$ = $\mathcal{P}_1$°($\mathcal{P}_2$°$\mathcal{P}_3$).

Theorem 6. Let App_1 and App_2 be two applications of a secure system with *STATES*$_0$ = *STATES*, and $\mathcal{P}_1$ = P_1 ∧ *Admin*(P_1) ∧ *Compat*(P_1, App_1), $\mathcal{P}_2$ = P_2 ∧ *Admin*(P_2) ∧ *Compat*(P_2, App_2) two SoD policies. If the applications App_1 and App_2 do not share roles, operations, or objects, then the policies $\mathcal{P}_1$ and $\mathcal{P}_2$ are composable.

In the following two examples we illustrate the use of the composition criterion. We show that not all SoD *policies* are composable despite the fact that their *SoD-P properties* are composable. For simplicity, we assume that the properties *Admin*(P_1), *Admin*(P_2), and *Admin*(P_1 ∧ P_2), where P_1 and P_2 are the two SoD-P properties, are satisfied in both examples.

Example 1. Let *poApp* = [{*purchase-order*}, {*read/write*, *sign*, *verify-sign*}, *poPlan*] be a "purchase-order processing" application, partitioned according to the plan *poPlan* = {(*purchase-order*, *read/write*), (*purchase-order*, *sign*), (*purchase-order*, *verify-sign*)}. The user assignments of this application allow a purchase order clerk, but no other user, to write and sign purchase orders, and purchase orders become valid documents only after they are countersigned by an approving authority. Within the RBAC system, the roles of the purchase-order clerk and that of the approval authority are separated and the users are assigned accordingly; i.e., the operations *read/write* and *sign* purchase orders are executed by a role r_1, and *verify-sign* purchase orders is executed by a role r_2. Furthermore, the purchase order clerk and the approval authority may not be the same user, and hence, different users are enrolled in r_1 and r_2 .

[2] We note that *Admin*(*SoD-P*) and *Compat*(*SoD-P, App*) are not properties of individual command sequences. Hence, they are neither "safety" nor "liveness" properties in the sense of Alpern and Schneider [2], and not subject to the Abadi-Lamport composition principle [1].

The following property specifies both the role assignments and the desired SoD property:

$poP =$

$(\forall s \text{ state of } \sigma, \forall r \in ROLES, r \neq r_1, r \neq r_2 \Rightarrow$

$execute \notin auth(s, r, poApp)) \wedge$

$(\forall s \text{ state of } \sigma, verify\_sign \notin auth(s, r_1, purchase\_order)$

$\wedge read / write \notin auth(s, r_2, purchase\_order) \wedge$

$sign \notin auth(s, r_2, purchase\_order)) \wedge$

$SSoD(\{r_1, r_2\}, poApp).$

Property *poP* is compatible with *poApp*, and consequently the policy $\mathcal{P}_1 = poP \wedge Compat(poP, poApp)$ is not empty. The administrator has to grant the permissions *read/write*, *sign* to the role r_1, and *verify-sign* to r_2, and to assign two different users to roles r_1 and r_2.

Let *apApp* = [{*purchase-order, check*}, {*read, read/write*, *sign*}, *apPlan*] be an "accounts payable" application, partitioned according to the plan *apPlan* = {(*purchase-order*, *read*), (*check*, *read/write*), (*check*, *sign*)}. The user assignments of this application allow a clerk, but no other user, to read purchase orders and write checks, but the checks must be signed by another user who has signature authority. Within the RBAC system, the roles of the clerk who writes checks and that of the check-signing authority are separated accordingly; i.e., the operations *read* purchase orders and *read/write* checks are executed by a role r_3, and *sign* checks are executed by a role r_4. Furthermore, the clerk and the signature authority must not be the same user, and hence, different users are enrolled in r_3 and r_4. The following property specifies both the role assignments and the desired SoD property:

$apP =$

$(\forall s \text{ state of } \sigma, \forall r \in ROLES, r \neq r_3, r \neq r_4 \Rightarrow$

$execute \notin auth(s, r, apApp)) \wedge$

$(\forall s \text{ state } \forall s \text{ state of } \sigma, sign \notin auth(s, r_3, check)) \wedge$

$(\forall s \text{ state of } \sigma, read \notin auth(s, r_4, purchase\_order)) \wedge$

$(\forall s \text{ state of } \sigma, read / write \notin auth(s, r_4, check)) \wedge$

$ObjDSoD(\{r_3, r_4\}, apApp).$

Property *apP* is compatible with *apApp*, and consequently the policy $\mathcal{P}_2 = apP \wedge Compat(apP, apApp)$ is not empty. The administrator has to grant the permissions *read* purchase orders, *read/write* checks to the role r_3, and *sign* checks to r_4, and assign two different users to roles r_3 and r_4.

Obviously, the policies $\mathcal{P}_1$ and $\mathcal{P}_2$ are composable under the composability criteria defined in this section: there are tranquil command sequences that execute both applications and satisfy both *poP* and *apP*.

Example 2. Let's define $\mathcal{P}_1$ as in Example 1, and let *cpApp* = [{*purchase-order*}, {*read/write*, *sign*}, *cpPlan*] be a "central purchasing" application, partitioned according to the plan is *cpPlan* = {(*purchase-order*, *read*), (*purchase-order*, *read/write*), (*purchase-order*, *sign*)}. The user assignments of this application require that the reading of the departmental purchase orders and the writing of organization purchase orders be separated from the signing of organization purchase order. Thus, in the RBAC system, the first two operations and the last operation of the plan are executed by separate roles r_1 and r_5. The following property specifies the role assignments and the desired SoD property:

$cpP =$

$(\forall s \text{ state of } \sigma, \forall r \in ROLES, r \neq r_1, r \neq r_5 \Rightarrow$

$execute \notin auth(s, r, cpApp)) \wedge$

$(\forall s \text{ state of } \sigma, read / write \notin auth(s, r_5, purchase\_order))$

$\wedge OpSSoD(\{r_1, r_5\}, cpApp).$

Property *cpP* is compatible with *cpApp*, and consequently the policy $\mathcal{P}_3 = cpP \wedge Compat(cpP, cpApp)$ is not empty. The administrator has only to grant *read/write* permissions to r_1 and *sign* permission to r_5 (and, of course, to have users assigned to these roles).

Policies $\mathcal{P}_1$ and $\mathcal{P}_3$ are *not* composable. To satisfy the *OpSSoD* property of policy $\mathcal{P}_3$, σ must start in a state such that r_1 does not have the *sign* permission. Such a tranquil command sequence could never execute *poApp*, because r_1 requires the *sign* permission by virtue of policy $\mathcal{P}_1$.

We note that the administrator can *redefine* policies $\mathcal{P}_1$ and $\mathcal{P}_3$ to obtain policies $\mathcal{P}_1$' and $\mathcal{P}_3$', which are composable. This can be done by making either one of the following changes via *non-tranquil* commands:

- add a new role in $\mathcal{P}_1$ and grant the *sign* permission to it and, at the same time, remove the *sign* permission from r_1; or
- replace r_1 in $\mathcal{P}_3$ with a new role that has only *read/write* permissions.

Note that, had we defined the above SoD policies only via SoD-P properties, all the policies would have appeared to be composable. However, Example 2 above shows that, when compatibility is taken into account, two of the SoD policies become incompatible and, hence, not composable.

6 Conclusions

The use of formalism in the definition of SoD properties helps (1) identify a wide variety of such properties, (2) remove ambiguities from informal definitions, and (3) establish heretofore unknown relationships among these properties. Formal SoD policies were also defined using the SoD properties.

The variety of SoD properties, while offering a wide choice of implementation, suggests that new

administrative methods and tools are necessary if these properties are to be effectively used. This is the case even for RBAC systems, which offer significant support for implementing SoD properties.

Acknowledgments

This paper was funded in part by the US Department of Commerce, National Oceanographic and Atmospheric Administration, under the SBIR Contract No. 50-DKNB-7-90120. The views expressed herein are those of the authors and do not necessarily reflect the views of the US Department of Commerce or any of its sub-agencies. The first author would like to thank C. Sekar Chandersekaran of IBM Corporation for his interest in, and support of, this work. We also thank John McLean for his helpful comments.

References

[1] Abadi M., and L. Lamport, "Composing specifications," *Stepwise refinement of Distributed Systems*, J. W. de Bakker, W. P. de Roever, and G. Rosenberg, eds., *Lecture Notes in Computer Science*, vol. 430, Springer-Verlag, 1990.

[2] Alpern B., and F. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, Oct. 1985, pp. 181-185.

[3] Clark D. D., and D. R. Wilson, "A Comparison of Commercial and Military Security Policies," Proc. of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, 1987, pp. 184-194.

[4] Clark D. D., and D. R. Wilson, "Evolution of a Model for Computer Integrity," in *Report of the Invitational Workshop on Data Integrity*, Z.G. Ruthberg and W.T. Polk (eds.), NIST Special Publication 500-168, Appendix A, September 1989.

[5] *Common Criteria for Information Technology Security Evaluation*, GISA, NNCSA, CESG, NIST, NSA, Version 2.0 Draft, December 1997.

[6] *Federal Criteria for Information Technology Security*, Vol. 1, Chapter 3 and Appendix C, Version 1.0, NIST and NSA, December 1992.

[7] Ferraiolo D., J. Cugini, and D. R. Kuhn, "Role-Based Access Control (RBAC): Features and Motivations," Proc. 1995 Computer Security Applications Conference, December 1995, pp. 241-248.

[8] Gligor, V. D., S. I. Gavrila, and J. Cugini, "The RBAC Security Policy Model", http://cspa09.ncsl.nist.gov/disk2/rbac/docs/model.ps

[9] Hecht, M. S., M. E. Carson, C. S. Chandersekaran, R. S. Chapman, L. J. Dotterer, V. D. Gligor, W. D. Jiang, A. Johri, G. L. Luckenbaugh, and N. Vasudevan, "Unix Without the Superuser," Proc. of the 1987 USENIX Conference, Phoenix, Arizona, June 1987, pp. 243-256.

[10] Hummel, A. A., K. Deinhart, S. Lorenz, V. D. Gligor, "Role-Based Security Administration," *Sicherheit in Informationsystemen* (K. Bauknecht, D. Karagiannis, and S. Teufel (eds.)), vdf Hochschulverlag, ETH Zurich, March 1996, pp. 69-92.

[11] Kailar R., V. D. Gligor, and L. Gong, "Security Effectiveness of Cryptographic Protocols," in *Dependable Computing for Critical Applications - 4*, F. Cristian, G. LeLann, and T. Lunt (eds.), Springer Verlag, 1995, pp. 139-157.

[12] Koch, G., and K. Loney, *Oracle. The complete reference*, Oracle Press, 1995.

[13] Nash M. J., and K. R. Poland, "Some Conundrums Concerning Separation of Duty," Proc. 1990 IEEE Symposium on Security and Privacy, Oakland, California, May 1990, pp. 201-207.

[14] Sandhu, R., "Transaction Control Expressions for Separation of Duties," Proc. of the 4th Aerospace Computer Security Conference, Tucson, Arizona, Dec. 1988, pp. 282-286.

[15] Simon R. T., and M. E. Zurko, "Separation of Duty in Role-Based Environments," Proc. of Computer Security Foundations Workshop X, Rockport, Massachusetts, June 1997.

[16] *Trusted Recovery Guideline,* NCSC-TG-022, Version 1, National Computer Security Center, December, 1989.

[17] *Unified INFOSEC Criteria*, INFOSEC Concepts, Section 3, "Dependencies among TCSEC Requirements (unclassified)", National Security Agency, 1993.

Appendix: Proofs

Proof of Theorem 1. We prove the theorem for the *SSoD* property. The proofs for the other properties are similar, and, for simplicity, are omitted. Note first that the *SSoD* property is a state property, and that all other state and transition properties are RBAC specific, and are satisfied in the secure RBAC system. Let σ be $op_1(s_0, S_1, obj_1, s_1)\cdot op_2(s_1, S_2, obj_2, s_2)\cdot\ldots$. We know that s_0 satisfies the *SSoD* property. Suppose that s_n , where $n\geq 0$, satisfies *SSoD*, and that s_{n+1} does not.. This implies that there are two roles r_1, r_2, sharing a common user u in state s_{n+1}. There are two possible cases:

a) r_1 and r_2 had the common user in state s_n;

b) the command $op(s_n, S_{n+1}, obj_{n+1}, s_{n+1})$ created or updated r_1 or r_2;

Case a) is rejected by the inductive hypothesis, and case b) is contradicted by σ being tranquil. Hence, s_{n+1} satisfies *SSoD*.

Proof of Theorem 2. Let σ be in *ObjDSoD*(*RoleSet, App*), and let σ start in s_0. When a user u performs two distinct operations, op_1, $op_2 \in OpSet$ on an object $obj \in ObjSet$, in roles of *RoleSet* and in states of σ, the system reaches a state s of σ such that $\{op_1, op_2\} \subseteq access\_history(s_0, s, u, RoleSet, obj)$. But *ObjDSoD* and the *access_history* properties imply that $access\_history(s_0, s, u, RoleSet, obj) \subseteq \{op_1\}$ and, hence, $\{op_1, op_2\} \subseteq \{op_1\}$,

contradiction. Conversely, assume, by way of contradiction, that $\sigma \notin ObjDSoD$. Then there is a state s in σ such that $access\_history(s_0, s, u, RoleSet, obj) \supseteq \{op_1, op_2\}$, where u is a user, $obj \in ObjSet$, op_1, $op_2 \in OpSet$. This means that u has performed two distinct operations on the same object of App.

Proof of Theorem 3. Similar to the proof of Theorem 2.

Proof of Theorem 4.

1. This is a consequence of the two SoD definitions.
2. The *1sSSSoD* property says that *RoleSet* has no roles with common role members, so that a *RoleSubset* in the *ObjSSoD* condition is reduced to a single role. Also by *ssSSoD*, that role is authorized to at most one operation of *OpSet*.
3. As before, the 1*sSSSoD* property reduces a *RoleSubset* with common role members to a single role, which is authorized to perform at most one operation of *OpSet*. Consequently, *OpSet* with a cardinality of at least 2 cannot be a subset of the authorized permissions of that single role.
4. This is a consequence of the two SoD definitions.
5. The roles enabled (active) for a user must be among those assigned to that user. Thus, if two roles have no common members, they cannot have common active users.
6. Let r be a role in *RoleSet*, $op_1 \neq op_2$ two operations in *OpSet*, and *obj* in *ObjSet*. If we apply the *ObjSSoD* property to $RoleSubset = \{r\}$, we get as a conclusion $\{op_1, op_2\} \not\subseteq \bigcup_{x \in RoleSubset} auth(s, x, obj) = auth(s, r, obj)$.
7. Assuming that *SSoD* holds, we have to prove the converse of 6. Let *RoleSubset* be a subset of roles in *RoleSet* with common members. *SSoD* implies that *RoleSubset* is reduced to a single role. For such *RoleSubset*, the *ObjSSoD* and *RObjSSoD* conditions coincide.
8. Let σ be a command sequence starting in s_0 and satisfying the *RObjSSoD* and *DSoD* properties, and $op(s_1, S, obj, s_2)$ a command of σ, with $r \in subject\_roles(s_1, S) \cap RoleSet \neq \emptyset$, $obj \in ObjSet$, $op \in OpSet$.Assume, by way of contradiction, that there is a role $r_1 \in RoleSet$ and an operation $op_1 \in OpSet$, $op_1 \neq op$, such that $op_1 \in access\_history(s_0, s_1, subject\_user(S), \{r_1\}, obj)$. Either $r = r_1$, and this means that in a state s of σ $\{op, op_1\} \subseteq auth(s, r, obj)$, which contradicts *RObjSSoD*, or $r \neq r_1$, and this means that $subject\_user(S)$ is active in both r and r_1 in a state of σ, which contradicts *DSoD*.
9. Let σ be a command sequence starting in s_0 and satisfying the *ObjSSoD* property, and $op(s_1, S, obj, s_2)$ a command of σ, with $r \in subject\_roles(s_1, S) \cap RoleSet \neq \emptyset$, $obj \in ObjSet$, $op \in OpSet$. Assume, by way of contradiction, that there is a role $r_1 \in RoleSet$ and an operation $op_1 \in OpSet$, $op_1 \neq op$, such that $op_1 \in access\_history(s_0, s_1, subject\_user(S), \{r_1\}, obj)$. We conclude that $subject\_user(S) \in role\_members(r) \cap role\_members(r_1)$, and $\{op, op_1\} \subseteq auth(s_1, r, obj) \cup auth(s_1, r_1, obj)$, which contradicts the *ObjSSoD* property.
10. Let σ be a command sequence satisfying the *OpSSoD* property. Let $|OpSet| \geq 2$ and suppose by way of contradiction that there is a role $r \in RoleSet$ and a state s of σ such that $OpSet \subseteq \bigcup_{obj \in ObjSet} auth(s, r, obj)$. Let us choose $RoleSubset = \{r\}$. The preceding inclusion can be written as $OpSet \subseteq \bigcup_{\substack{x \in RoleSubset \\ obj \in ObjSet}} auth(s, x, obj)$, which contradicts the *OpSSoD* property.
11. If σ satisfies the *SSoD* property, any *RoleSubset* with common members is reduced to a single role, and for such subsets the *OpSSoD* and *ROpSSoD* conditions are equivalent.
12. Let σ be a command sequence that satisfies the *DSoD* and *ROpSSoD* properties. Because of *DSoD*, any *RoleSubset* with common active users (as required by the *OpDSoD* property) is reduced to a single role, $RoleSubset = \{r\}$. The *OpDSOD* condition $OpSet \not\subseteq \bigcup_{\substack{x \in RoleSubset \\ obj \in ObjSet}} auth(s, x, obj)$ may be written $OpSet \not\subseteq \bigcup_{obj \in ObjSet} auth(s, r, obj)$, which is true by the *ROpSSoD* property.
13. Let σ be a command sequence that satisfies the *OpSSoD* property, and *RoleSubset* a subset of *RoleSet* with common active users. Then *RoleSubset* also has common members. If we assume $|OpSet| \geq 2$ and apply *OpSSoD*, we get exactly the conclusion of *OpDSoD*.
14. Let σ be a command sequence that starts in s_0 and satisfies the *ObjDSoD* property, $|OpSet| \geq 2$, and $op(s_1, S, obj, s_2)$ a command in σ with $op \in OpSet$, $obj \in ObjSet$, $r \in subject\_roles(S) \cap RoleSet$. Assume, by way of contradiction that $OpSet \subseteq access\_history(s_0, s_1, u, RoleSet, obj) \cup \{op\}$. There must be $op_1 \in OpSet$, $op_1 \neq op$, and $r_1 \in RoleSet$, such that $op_1 \in access\_history(s_0, s_1, subject\_user(S), \{r_1\}, obj)$, which contradicts the *ObjDSoD* property.
15. Let σ be a command sequence that starts in s_0 and satisfies the *OpDSoD* property, $|OpSet| \geq 2$, and $op(s_1, S, obj, s_2)$ a command of σ such that $op \in OpSet$, $obj \in ObjSet$, $r \in subject\_roles(S) \cap RoleSet$. Suppose by way of contradiction that $OpSet \subseteq access\_history(s_0, s_1, subject\_user(S), RoleSet, obj) \cup \{op\}$. But $access\_history(s_0, s_1, subject\_user(S), RoleSet, obj) \subseteq \bigcup_{x \in RoleSet} auth(s_1, x, obj)$, and $op \in auth(s_1, r, obj)$.

Consequently, $OpSet \subseteq \bigcup_{x \in RoleSet} auth(s_1, x, obj)$, which contradicts the *OpDSoD* property.

Proof of Theorem 5.

(1) Let P_1 and P_2 be two SoD properties. If we can find a start state s_0 satisfying the state invariants of both P_1 and P_2, then clearly $\hat{s}_0 \in P_1 \cap P_2$. We note that if the roles assigned to an application satisfy a SoD property P, then after revoking a permission from one of the roles or removing a member from one of the roles, they still satisfy P. Now, starting in an arbitrary state with the current roles, and through possibly non-tranquil commands, we can revoke permissions one by one and/or remove members one by one from the App_1's roles until we reach a state that satisfies P_1. It is always possible to do that, because the empty role (without permissions) with no members satisfies any SoD property. Then, we can apply the same procedure to App_2's roles, until we reach a state that satisfies P_2 (and P_1). This new state, which satisfies both P_1 and P_2, is the new start state.

(2) Let App_1=[{obj_1}, {op_1, op_2}, $plan_1$], App_2=[{obj_2}, {op_3, op_4}, $plan_2$] be two applications in a secure RBAC system, and $RoleSet_1$={r_1, r_2}, $RoleSet_2$ = {r_3, r_4} their assigned roles. Let $STATES_0$ = {s_0', s_0''}, with the states s_0', s_0'' such that:

- in s_0', r_1 and r_2 do not have common assigned users, and r_3 has permissions op_3, op_4; and
- in s_0'', r_1 and r_2 have a common assigned user, and r_3 has only permission op_3, and r_4 has only permission op_4.

Then, P_1=*SSoD*($RoleSet_1$, App_1)≠∅, P_2= *ROpSSoD* ($RoleSet_2$, App_2) ≠∅, but $P_1 \cap P_2 = \emptyset$. Indeed, any tranquil command sequence starting in s_0' satisfies P_1 but not P_2, and any tranquil command sequence starting in s_0'' satisfies P_2 but not P_1.

(3) P_1=*SSoD*($RoleSet_1$, App_1)≠∅ and P_2= *ROpSSoD* ($RoleSet_2$, App_2) ≠∅ means that the start state s_0 satisfies $P_1 \wedge P_2$. At least $\hat{s}_0 \in P_1 \cap P_2$.

Proof of Theorem 6. Suppose that $\exists \sigma_1 \in P_1$: σ_1 executes $App_1 \wedge \exists \sigma_2 \in P_2$: σ_2 executes App_2. Hence, there are start states s_0' satisfying P_1 and s_0'' satisfying P_2. Then, starting from s_0', for example, and using possibly non-tranquil transitions, we can reach a state s_0 satisfying both P_1 and P_2 (their constraints do not interfere). Taking s_0 as start state, we can apply the commands of σ_1 executing App_1, then the commands of σ_1 executing App_2, which again do not interfere.