Constraint Generation for Separation of Duty

Hong Chen  
Dept. of Computer Science  
Purdue University  
chen131@cs.purdue.edu

Ninghui Li  
Dept. of Computer Science  
Purdue University  
inghui@cs.purdue.edu

ABSTRACT
Separation of Duty (SoD) is widely recognized to be a fundamental principle in computer security. A Static SoD (SSoD) policy states that in order to have all permissions necessary to complete a sensitive task, the cooperation of at least a certain number of users is required. In Role-Based Access Control (RBAC), Statically Mutually Exclusive Role (SMER) constraints are used to enforce SSoD policies. This paper studies the problem of generating sets of constraints that (a) enforce a set of SSoD policies, (b) are compatible with the existing role hierarchy, and (c) are minimal in the sense that there is no other constraint set that is less restrictive and satisfies (a) and (b).

Categories and Subject Descriptors

General Terms
Algorithms, Security

Keywords
role based access control, separation of duty, constraints

1. INTRODUCTION
Separation of Duty (SoD) is widely recognized as a fundamental principle in computer security [4, 21]. In its simplest form, the principle states that a sensitive task should be performed by two different users acting in cooperation. The concept of SoD has long existed before the information age; it has been widely used in, for example, the banking industry and the military, sometimes under the name “the two-man rule”. More generally, an SoD policy requires the cooperation of at least \( k \) different users to complete the task. To ensure that at least \( k \) different users are involved to complete a task, one approach is to require that no \( k - 1 \) users together have all the permissions needed to complete the task. We call such a requirement a Static Separation of Duty (SSoD) Policy, following the terminology in [17]. In Role-Based Access Control (RBAC) [3, 8, 9, 10, 11, 24], SSoD policies are usually enforced using constraints that restrict the role memberships of users. For example, one constraint may declare two roles \( r_1 \) and \( r_2 \) to be mutually exclusive, in the sense that no user is allowed to be a member of both \( r_1 \) and \( r_2 \). More generally, a constraint may require that no user is a member of \( t \) or more roles in a set of \( m \) roles \( \{r_1, r_2, \ldots, r_m\} \). We call these constraints Statically Mutually Exclusive Role (SMER) constraints. SMER constraints are part of most RBAC models, including the RBAC96 models by Sandhu et al. [24] and the proposed and adopted ANSI/NIST standard for RBAC [3, 11], in which SMER constraints are referred to as SSD constraints.

SSoD policies should not be equated with SMER constraints. Each SSoD policy specifies the minimum required number of users that are allowed to together possess all permissions for a sensitive task. Such a policy can be specified independent of whether roles are used to manage the permissions or not. On the other hand, SMER constraints are specific to RBAC. Each constraint limits the role memberships a single user is allowed to have. Whether a set of SMER constraints is sufficient to enforce a given SSoD policy depends upon how permissions are assigned to roles. For example, if all permissions that are needed to complete a sensitive task are assigned to a single junior-most role, one cannot use SMER constraints to ensure that no single user possesses all the permissions, because no SMER constraint can prevent a user from being assigned to that single role and thereby gaining all permissions needed for the task.

Li et al. [17] studied the relationship between SSoD policies and SMER constraints. The following two problems were introduced: the verification problem asks “does a set of SMER constraints enforce an SSoD policy?”; and the generation problem asks “how to generate a set of SMER constraints that is adequate to enforce an SSoD policy?”. Li et al. showed that the verification problem is intractable (coNP-complete). They also noted that often multiple sets of SMER constraints can enforce an SSoD policy, and these constraint sets have different degree of restrictiveness. Intuitively, when two sets of constraints all enforce the desirable SSoD policy, one would prefer the set that is less restrictive. Li et al. introduced the notion that a set \( C \) of SMER constraints minimally enforces a set \( E \) of SSoD policies, which means that \( C \) enforces \( E \) and there does not exist any other set of constraints that is both less restrictive than \( C \) and also enforces \( E \). Li et al. presented an algorithm that generates singleton sets of SMER constraints that minimally enforce the policies.

The constraint generation work in [17] does not consider the interaction between role hierarchies and constraints. More specifically, the algorithm in [17] may generate SMER constraints that preclude any user from being authorized for some roles in a role...
hierarchy. For example, if \( \{ (r_3 \geq r_1), (r_3 \geq r_2) \} \subseteq RH \), then the constraint \( smer(\{r_1, r_2\}, 2) \) implies that no user is allowed to be authorized for \( r_3 \). This is undesirable, because, if no user is allowed to be authorized for a role, then there is no reason in having that role as part of the role hierarchy. Another limitation of the work in [17] is that it generates only singleton constraint sets.

In this paper, we study constraint generation while considering the impact of role hierarchies. We define the notion of compatibility between a role hierarchy and a set of SMER constraints, and then present necessary and sufficient conditions for compatibility. Recall that each SMER constraint requires that no user is a member of \( f \) or more roles in a set of \( m \) roles. We show that, for any integer \( j > 2 \), SMER constraints with \( t = j \) provide additional expressive power than using only SMER constraints with \( t < j \). We then study the problem of generating constraint sets that are compatible with the given role hierarchy, enforce the desired SSoD policies, and are minimal. We present two algorithms for generating such constraint sets.

The rest of this paper is organized as follows. We discuss related work in Section 2, and give preliminary definitions in Section 3. In Section 4, we study the notion of compatibility, and discuss the expressive power of SMER constraints. In Section 5, we discuss how to generate SMER constraints to implement SSoD policies, and give two algorithms. We also discuss experimental results of the algorithms in Section 6. We conclude this paper with Section 7.

2. RELATED WORK

To our knowledge, in the information security literature the notion of SoD first appeared in Saltzer and Schroeder [21] under the name “separation of privilege.”

Clark and Wilson’s commercial security policy for integrity [4] identified SoD along with well-formed transactions as two major mechanisms for controlling fraud and error. The use of well-formed transactions ensures that information within the computer system is internally consistent. Separation of duty ensures that the objects in the physical world are consistent with the information about these objects in the computer system. As Clark and Wilson [4] explained: “Because computers do not normally have direct sensors to monitor the real world, computers cannot verify external consistency directly. Rather, the correspondence is ensured indirectly by separating all operations into several subparts and requiring that each subpart be executed by a different person.”

Sandhu [22, 23] presented Transaction Control Expressions, a history-based mechanism for dynamically enforcing SoD policies. Nash and Poland [19] explained the difference between dynamic and static enforcement of SoD policies. In the former, a user may perform any step in a sensitive task provided that the user does not also perform another step on that data item. In the latter, users are constrained a-priori from performing certain steps.

There exists a wealth of literature [1, 2, 6, 13, 14, 15, 25, 26] on constraints other than SMER constraints in RBAC. They either proposed and classified new kinds of constraints [13, 25] or proposed new languages for specifying sophisticated constraints [1, 2, 6, 15, 26]. Most of the proposed constraints are variants of SMER constraints; for example, one may declare that two permissions are mutually exclusive, so that no role can be authorized for both permissions, or that two roles are dynamically mutually exclusive, so that they cannot be activated in the same session.

Kuhn [16] discussed mutual exclusion of roles for separation of duty and proposed a safety condition: that no one user should possess the privilege to execute every step of a task, thereby being able to complete the task.

Crampton [5] discussed a set-based approach to separation of duty: the state of the access control system is defined as a set of sets, and a constraint is defined as a set which should be forbidden in the system state. The system state satisfies the constraint if no element of the system state (which is a set) is a superset of the constraint. The comparison of restrictiveness between different constraints is discussed.

As discussed in Section 1, our work is directly motivated by the work by Li et al. [17]. See Section 1 for a discussion of [17].

3. PRELIMINARY DEFINITIONS

We now reproduce the definitions of SSoD policies and SMER constraints from [17]. We assume that there are three countably infinite sets: \( U \) (the set of all possible users), \( R \) (the set of all possible roles), and \( P \) (the set of all possible permissions).

Definition 1 (SSoD Policy). A \( k \)-n SSoD (\( k \)-out-of-\( n \) Static Separation of Duty) policy is expressed as \( ssod(\{p_1, \ldots, p_n\}, k) \)

where \( \{p_1, \ldots, p_n\} \subseteq P \) is a set of permissions and \( n \) and \( k \) are integers such that \( 1 < k \leq n \). This policy means that there should not exist a set of fewer than \( k \) users that together have all the permissions in \( \{p_1, \ldots, p_n\} \). In other words, at least \( k \) users are required to perform a task that needs all these permissions.

Definition 2 (RBAC state). An RBAC state \( \gamma \) is a 3-tuple \( (UA, PA, RH) \), in which the user assignment relation \( UA \subseteq U \times R \) associates users with roles, the permission assignment relation \( PA \subseteq R \times P \) associates roles with permissions, and the role hierarchy relation \( RH \subseteq R \times R \) specifies an acyclic relation among roles.

The reflexive, transitive closure of \( RH \) (denoted by \( RH^* \)) is a partial order among roles in \( R \). When \( (r_1, r_2) \in RH^* \), we write \( r_1 \geq_{RH} r_2 \) and say that \( r_1 \) is senior to \( r_2 \) (or, equivalently, \( r_2 \) is junior to \( r_1 \)).

An RBAC state \( \gamma = (UA, PA, RH) \) determines the set of roles of which each user is a member, and the set of permissions for which each user is authorized. Formally, \( \gamma \) is associated with two functions, \( auth \_roles \_gamma : U \to 2^R \) and \( auth \_perms \_gamma : U \to 2^P \). The two functions are defined as follows:

\[
\begin{align*}
auth \_roles \_gamma (UA, PA, RH) [u] &= \{ r \in R \mid \exists r_1 \in R \{ (u, r_1) \in UA \land r_1 \geq_{RH} r \} \} \\
auth \_perms \_gamma (UA, PA, RH) [u] &= \{ p \in P \mid \exists r_1, r_2 \in R \{ (u, r_1) \in UA \land r_1 \geq_{RH} r_2 \land (r_2, p) \in PA \} \}
\end{align*}
\]

As \( auth \_roles \_gamma (UA, PA, RH) [u] \) is determined only by \( UA \) and \( RH \), we sometimes write \( auth \_roles \_gamma (UA, RH) [u] \).

Definition 3 (SSoD Safety). We say that an RBAC state \( \gamma \) is safe with respect to an SSoD policy \( ssod(\{p_1, \ldots, p_n\}, k) \) if in state \( \gamma \) no \( k-1 \) users together have all the permissions in the policy. More precisely,

\[
\forall u_1 \cdots u_{k-1} \in U \frac{\bigcup_{i=1}^{k-1} auth \_perms \_gamma [u_i]}{\not\in} \{p_1, \ldots, p_n\}.
\]

An RBAC state \( \gamma \) is safe with respect to a set \( E \) of SSoD policies if it is safe with respect to every policy in the set, and we write this as \( safe_E(\gamma) \).
We use a running example to illustrate the concepts in this paper. This example is shown in Figure 1 and is explained below.

**Example 1.** Consider the following singleton set of SSoD policies:

\[ E = \{ \text{ssod}\{\{p_1, p_2, p_3, p_4\}, 2\} \} \]

which means that at least two users are required to have all permissions in \( \{p_1, p_2, p_3, p_4\} \). Consider the following three RBAC states \( \gamma_1 = \langle UA_1, PA, RH \rangle \), \( \gamma_2 = \langle UA_2, PA, RH \rangle \), and \( \gamma_3 = \langle UA_3, PA, RH \rangle \), in which \( PA, RH, UA_1, UA_2, UA_3 \) are defined in Figure 1. These three states have the same \( PA \) and \( RH \), and differ only in \( UA \). The state \( \gamma_1 \) is safe with respect to \( E \); but the states \( \gamma_2 \) and \( \gamma_3 \) are not, because in these two states, the user \( u_1 \) has all permissions in \( \{p_1, p_2, p_3, p_4\} \).

**Definition 4 (SMER constraint).** A \( t \)-in-\( m \) SMER (t-out-of-\( m \) Statically Mutually Exclusive Role) constraint is expressed as

\[ \text{smer}\{\{r_1, \ldots, r_m\}, t\} \]

where \( \{r_1, \ldots, r_m\} \) is a set of roles, and \( m \) and \( t \) are integers such that \( 1 < t \leq m \). This constraint forbids a user from being a member of \( t \) or more roles in \( \{r_1, \ldots, r_m\} \).

A \( t \)-\( m \) SMER constraint is said to be canonical of cardinality \( t \) when \( t = m \).

In [17] it has been shown that each \( t \)-\( m \) SMER (\( t \leq m \) constraint can be equivalently encoded as a set of \( t \)-\( t \) SMER constraints. Thus in this paper, we treat a \( t \)-\( m \) SMER constraint as a set of \( t \)-\( t \) SMER (canonical) constraints.

**Figure 1:** A running example. The role hierarchy and permission assignment are shown in the picture. Roles are shown in solid boxes, and permissions in dashed boxes. A solid line segment represents a role-role relationship, and a dash line segment represents the assignment of a permission to a role.

\[ E = \{ \text{ssod}\{\{p_1, p_2, p_3, p_4\}, 2\} \} \]

\[ PA = \{ (r_1, p_1), (r_2, p_2), (r_3, p_3), (r_4, p_3), (r_5, p_1) \} \]

\[ RH = \{ (r_4 \geq r_1), (r_4 \geq r_2) \} \]

\[ UA_1 = \{ (u_1, r_1), (u_1, r_3), (u_1, r_5) \} \]

\[ UA_2 = \{ (u_1, r_3), (u_1, r_4) \} \]

\[ UA_3 = \{ (u_1, r_1), (u_2, r_2), (u_1, r_3) \} \]

\[ C_1 = \{ \text{smer}\{\{r_1, r_2, r_3\}, 3\}, \text{smer}\{\{r_1, r_2, r_4, r_5\}, 4\} \} \]

\[ C_2 = \{ \text{smer}\{\{r_3, r_4\}, 2\}, \text{smer}\{\{r_1, r_2, r_5\}, 3\} \} \]

\[ C_3 = \{ \text{smer}\{\{r_1, r_3\}, 2\}, \text{smer}\{\{r_2, r_5\}, 2\} \} \]

\[ C_4 = \{ \text{smer}\{\{r_1, r_2\}, 2\} \} \]

The SSoD policy we want to enforce

\[ \text{The permission assignment relation.} \]

\[ \text{The role hierarchy.} \]

\[ \text{A safe user assignment relation} \]

\[ \text{An unsafe user assignment relation; } u_1 \text{ has all permissions} \]

\[ \text{An unsafe user assignment relation; } u_1 \text{ has all permissions} \]

\[ C_1 \text{ enforces } E. \text{ Both } UA_2 \text{ and } UA_3 \text{ violate } C_1 \text{ because } \]

\[ u_1 \text{ is authorized for } \{r_1, r_2, r_3\} \].

\[ C_2 \text{ does not enforce } E. \text{ While } UA_2 \text{ violates } E, UA_3, \text{ which is unsafe wrt. } E, \text{ satisfies } C_2. \]

\[ C_3 \text{ enforces } E; \text{ however, it is too restrictive, as it rules out } UA_1. \]

\[ C_4 \text{ enforces } E; \text{ however, } C_4 \text{ is incompatible with } RH, \text{ as no user can be assigned to } r_4. \]

**Definition 5 (SMER Satisfaction).** We say that an RBAC state \( \gamma \) satisfies a \( t \)-\( m \) SMER constraint \( \text{smer}\{\{r_1, \ldots, r_m\}, t\} \) when

\[ \forall u \in U \quad (| \text{auth}\_\text{roles}_\gamma[u] \cap \{r_1, \ldots, r_m\}| < t). \]

If \( \gamma \) does not satisfy a SMER constraint, we say that \( \gamma \) violates the SMER constraint. An RBAC state satisfies a set \( C \) of SMER constraints if it satisfies every constraint in the set, and we write this as \( \text{satisfies}_C(\gamma) \).

Observe that a SMER constraint is concerned with only the role membership of each user, which does not depend on \( PA \); thus we sometimes say that \( UA \) and \( RH \) satisfy the SMER constraints and write \( \text{satisfies}_C(UA, RH) \).

SMER constraints restrict the role memberships each individual user is allowed to have. Provided that permissions are carefully assigned to roles, SMER constraints restrict the permissions each individual user is allowed to have. When SMER constraints are motivated by SSoD policies, we need to ensure that SMER constraints place sufficient restrictions on each individual user so that any set of \( k - 1 \) users do not have all permissions to complete a sensitive task. This means that, no matter how users are assigned to roles, as long as the SMER constraints are satisfied, we want to

**Definition 6 (SSoD enforcement).** An SSoD configuration is a 3-tuple \( \langle E, PA, RH \rangle \), where \( E \) is a set of SSoD policies, \( PA \) is a permission assignment relation, and \( RH \) is a role hierarchy.

We say that a set \( C \) of SMER constraints enforces the SSoD configuration \( \langle E, PA, RH \rangle \) if and only if
∀UA ⊂ U × R [satisfiesC((UA, PA, RH))] 
⇒ safeE((UA, PA, RH))

In other words, if C enforces (E, PA, RH), then C rules out every user-role assignment relation UA such that ⟨UA, PA, RH⟩ is not safe with respect to E.

Example 2. Continuing the example in Figure 1. Consider the three sets of constraints C₁, C₂, and C₃. C₁ enforces (E, PA, RH). The constraints in C₁ require that no user is authorized for all the of r₂, r₃, and r₄, or all of r₁, r₂, r₄, and r₅. Any role combination that enables a user to have all permissions in {p₁, p₂, p₃, p₄} violates one of the constraints in C₁.

As we discuss in Example 1, both γ₂ = (UA₂, PA, RH) and γ₃ = (UA₃, PA, RH) are unsafe with respect to E. In both γ₂ and γ₃, u₁ is authorized for {r₁, r₂, r₃}, thus violating C₁.

C₂ does not enforce (E, PA, RH), because it does not rule out all UAs that are unsafe. For example, γ₃ = (UA₃, PA, RH) is unsafe w.r.t. E, but γ₃ does not violate C₂ because auth_roles{u₁} = {r₁, r₂, r₃}; thus, |auth_roles{u₁} ∩ {r₁, r₄}| = 1 < 2, and |auth_roles{u₁} ∩ {r₁, r₂, r₃}| = 2 < 3.

C₃ enforces (E, PA, RH), as it is more restrictive than C₁. In fact, it is more restrictive than necessary; for example, γ₁ = (UA₁, PA, RH) is safe w.r.t. E; however, γ₁ violates C₃ because u₁ is authorized for both r₁ and r₂.

Note that not all SSoD configurations can be enforced by SMER constraints. For example, given an SSoD configuration ((ssod(P, k), PA, RH = ∅) such that all permissions in P are assigned to one role r, then no set of constraints can prevent from a user being assigned to r; thus, the SSoD configuration is not enforceable.

4. COMPATIBILITY & IMPLEMENTABILITY

In [17], Li et al. showed that any enforceable SSoD configuration can be enforced by using only 2-2 SMER constraints. Given an enforceable SSoD configuration (E, PA, RH), one can declare every pair of roles in Roles[RH] ∪ Roles[PA] to be mutually exclusive, thereby enforcing E. However, this naive strategy often results in constraints that are so restrictive that they render some roles in the role hierarchy useless. More precisely, a set of SMER constraints may preclude one from assigning any user to some roles in RH. Consider the example in Figure 1, the constraint smer({r₁, r₂}, 2) implies that no user is allowed to be authorized for r₄. Note that this means that no user can be assigned to r₄, or r₅ (which is not present in Figure 1). This is undesirable, because, if no user is allowed to be authorized for a role, then there is no reason for having that role as part of the role hierarchy. To address this, we define the notion of compatibility between a set of SMER constraints and RH, and then present necessary and sufficient conditions for it.

4.1 Compatibility

Definition 7 (Compatibility and Incompatibility). We say that a set C of SMER constraints is incompatible with a role hierarchy RH, if and only if there is a role r ∈ Roles[RH] such that for any user assignment relation UA which satisfies C under RH, no user is authorized for r. C is compatible with RH if and only if C is not incompatible with RH.

The above definition is based on the intuition that every role must be “usable” in some state that satisfies the constraints in C. We now study how to determine whether a set of SMER constraints is compatible with a role hierarchy. Following is a necessary and sufficient condition for a set of SMER constraints to be compatible with a role hierarchy.

Lemma 1. A set C of SMER constraints RH is incompatible with a role hierarchy if and only if there is a SMER constraint c = smer(R, t) ∈ C such that t roles in R share a common ancestor in RH.

Proof. For the “if” part, suppose there is a SMER constraint c = smer(R, t) ∈ C and R contains a subset R′, |R′| = t and all roles of R′ share a common ancestor r in RH. Then any user assignment UA which satisfies C will have no user authorized for r, because if any user is authorized for r, c is violated. Therefore C is incompatible with RH.

For the “only if” part, suppose C is incompatible with RH. Let r be the “unsuitable” role, any user assignment which satisfies C will have no user authorized for r. Consider an user assignment UA which contains only one user u and u is assigned with r. By definition of incompatibility UA doesn’t satisfy C. Suppose UA violates c = smer(R, t) ∈ C. Then u must be authorized for some t roles in R, and those roles share the same ancestor r.

The above lemma tells us that one can efficiently check whether C is compatible with a set RH of constraints. For every constraint c = smer(R, t) ∈ C and every role r in RH, let R be the intersection of R and all the roles junior to r (including r itself). If for some c and r, |R′| ≥ t, then C is incompatible with RH. Otherwise C is compatible with RH.

4.2 Implementing SSoD policies using SMER constraints

We now introduce the notion of a set C of SMER constraints implements an SSoD configuration (E, PA, RH), which requires that C enforces (E, PA, RH), while being compatible with RH.

Definition 8 (Implementing SSoD policies). Given PA ⊂ R × P, RH ⊂ R × R, a set E of SSoD policies, and a set C of SMER constraints. We say C implements E (under PA and RH) when

1. C is compatible with RH.
2. C enforces E under PA and RH, i.e.,

∀UA ⊂ U × R [satisfiesC((UA, PA, RH))] 
⇒ safeE((UA, PA, RH))

In Figure 1, the constraint set C₄ enforces E under PA and RH; however, C₄ is incompatible with RH, as it means no user can be authorized for r₄. Thus, C₄ does not implement E.

Definition 9 (Implementable SSoD configurations). An SSoD configuration is a 3-tuple (E, PA, RH), where PA is a permission assignment relation, RH is a role hierarchy, and E is a set of SSoD policies. An SSoD configuration is implementable if there exists a set C of SMER constraints such that C implements E under PA and RH.

Lemma 2. An SSoD configuration (E, PA, RH) is not implementable if and only if there exists an SSoD policy ssod({p₁, · · · , pₙ}, k) in E such that k − 1 roles together have all the permissions in {p₁, · · · , pₙ}.

Proof. For the “if” part, assume that there exists such an SSoD policy. Then no matter what set of SMER constraints we use, either the set is incompatible with RH, or one can assign k − 1 different users to the k − 1 roles without violating any constraint from the set, resulting in an unsafe state. In either case, the set of SMER constraints does not enforce E.
For the “only if” part, assume that there does not exist such an SSoD policy. We now need to show that there exists a set \( C \) of SMER constraints that enforces \( E \). We now construct such a set \( C \) of constraints. Our approach is to generate the most restrictive set of constraints (the formal definition of “restrictiveness” is in Section 5.2). For example, if \( RH \) is empty, then we can declare every pair of roles to be mutually exclusive, this would enforce \( E \). However, when \( RH \) is not empty, then we need to ensure that \( C \) is compatible with \( RH \). For example, when two roles \( r_1 \) and \( r_2 \) have a common ancestor in \( RH \), then these two roles cannot be declared to be mutually exclusive.

Construct \( C \) as follows. We begin with \( C = \emptyset \). Let \( R \) be the set of all roles occurring in \( PA \) or \( RH \). For each nonempty subset \( S \) of \( R \) such that all roles in \( S \) have a common ancestor (each singleton set would satisfy the condition), for every \( r \in R \) such that the set \( S \cup \{r\} \) does not have a common ancestor, add \( \text{smer}(S \cup \{r\}, |S| + 1) \) to \( C \). The same \( \text{smer} \) constraint may be added more than once; as \( C \) is a set, the duplicate ones are ignored.

We first observe that, from Lemma 1, \( C \) is compatible with \( RH \), because for every \( t \)-m SMER constraint in \( C \), \( t = m \) and the \( m \) roles in the constraint do not share a common ancestor. Suppose, for the sake of contradiction, that \( C \) does not enforce \( E \), then there exists in \( E \) an SSoD policy \( \text{ssod}(\{p_1, \ldots, p_n\}, k) \) and \( UA \) such that \( k \) users together have all permissions in \( \{p_1, \ldots, p_n\} \) without violating any constraint in \( C \). Let \( R_1, \ldots, R_k \) be the role memberships of the \( k \)-1 users. For each \( R_i \), all roles in it must share a common ancestor. The reason is that if these roles do not, then let \( S \subseteq R \) be a largest subset of \( R_i \) that shares a common ancestor and \( r \) be any role in \( R_i - S \), there exists a constraint \( \text{smer}(S \cup \{r\}, |S| + 1) \in C \); this constraint is violated. Let \( r_j \) be a common ancestor \( R_j \), then the \( k-1 \) roles \( \{r_1, \ldots, r_{k-1}\} \) together have all permissions in \( \{p_1, \ldots, p_n\} \), contradicting the assumption that such situation does not exist.

**Theorem 3.** Checking whether an SSoD configuration is enforceable is coNP-complete.

**Proof.** The proof is similar to the one in [17] for the theorem that checking whether an RBAC state is safe or not wrt. a set if SSoD policies is coNP-complete. We first show that determining that an SSoD configuration is not enforceable is in NP. If an SSoD configuration is not enforceable, according to Lemma 2, there must exist an SSoD policy \( \text{ssod}(\{p_1, \ldots, p_n\}, k) \) in \( E \) such that \( k \) roles together have all the permissions in \( \{p_1, \ldots, p_n\} \). After such a policy and the \( k - 1 \) roles are guessed, verifying that these roles indeed have all the permissions takes polynomial time.

We now show that determining whether an SSoD configuration is not enforceable is NP-hard by reducing the set covering problem to it. In the set covering problem, the inputs are a finite set \( S \), a family \( F = \{S_1, \ldots, S_t\} \) of subsets of \( S \), and a budget \( B \). The goal is to determine whether there exist \( B \) sets in \( F \) whose union is \( S \). This problem is NP-complete [12, 20]. The reduction is as follows. Given \( S, F, B \), construct an SSoD policy \( E \) as follows: For each element in \( S \), we create a permission for it, let \( k \) be \( B + 1 \) and let \( n \) be the size of \( S \). We have constructed a \( k-n \) SSoD policy \( \text{ssod}(S, B + 1) \). Construct \( PA \) and \( RH \) as follows. For each different subset \( S_i \) (1 \( \leq i \leq t \)) in \( F \), create a new role \( r_i \) and assigns to it the permissions corresponding to the elements in \( S_i \). The resulting SSoD configuration is not enforceable if and only if \( B \) sets in \( F \) cover \( S \).

### 4.3 Expressive power

In [17], it was shown that any enforceable SSoD configuration can be enforced using only 2-2 SMER constraints, even though this may result in constraints that are more restrictive than necessary. We now show if we require constraints to be compatible with the given role hierarchy, t-t SMER constraints for each larger \( t \) adds new expressive power in terms of implementing SSoD configurations. Since \( t \)-m SMER has the same expressive power as a set of \( t \)-t SMER constraints, we can also see that \( t \)-m SMER constraints for each larger \( t \) also adds new expressive power.

In Figure 2, the policy \( E \) says no single user can possess all permissions in \( \{p_1, p_2, p_3\} \). The permission assignment relation \( PA \) is such that \( p_1, p_2, \) and \( p_3 \) are assigned to \( r_1, r_2, \) and \( r_3 \), respectively. The role hierarchy \( RH \) is such that any two of \( r_1, r_2, \) and \( r_3 \) have a common ancestor. The 3-3 SMER constraint \( c \) implements \( E \) under \( RH \). However using 2-2 SMER constraints alone, one cannot implement \( E \), because any such constraint will be incompatible with \( RH \). More generally, we have the following theorem.

**Theorem 4.** For any integer \( t > 2 \), there exists a SSoD configuration that cannot be implemented using canonical constraints of cardinality less than \( t \), but can be implemented using canonical constraints of cardinality \( t \).

**Proof.** Given any integer \( t > 2 \), consider the following configuration with \( 2t \) roles and \( t \) permissions:

\[
PA = \{(r_1, p_1), \ldots, (r_t, p_t)\} \\
RH = \{(r_i \geq r_j) \mid 1 \leq i, j \leq t \land i \neq j\} \\
E = \{\text{ssod}(\{p_1, \ldots, p_t\}, 2)\}
\]

In this configuration, every role in \( \{r_1, \ldots, r_t\} \) is associated with one permission. And every \( t - 1 \) roles of \( \{r_1, \ldots, r_t\} \) have a common ancestor. The policy says no single user should acquire all the permissions.

Given any set \( C \) of canonical SMER constraints that implements the configuration, \( C \) is violated by the user assignment \( UA = \{(u_1, r_1), \ldots, (u_t, r_t)\} \) since \( UA \) is not safe with respect to \( E \). Note that \( \text{auth}_t(\text{roles}(UA, RH))|u_i| = \{r_1, r_2, \ldots, r_t\} \). Now consider any \( c = \text{smer}(R, p) \in C \) that is violated by \( UA \). We have \( |R| = p \), since \( c \) is canonical. We also have \( R \subseteq \{r_1, \ldots, r_t\} \), because otherwise \( c \) will not be violated. Finally, we have that \( p \) should not be less than \( t \), since otherwise \( c \) will be incompatible with \( RH \). Therefore, \( R = \{r_1, r_2, \ldots, r_t\} \).

We have shown that every set of canonical SMER constraints that implements the configuration must include the \( t-t \) SMER constraint \( \text{smer}(\{r_1, r_2, \ldots, r_t\}, t) \). It thus follows that any set of constraints that contains only canonical constraints of size less than
t does not implement the configuration. And we can also see that \{smert\{r_1, r_2, \ldots, r_n\}, t\} implements the configuration, since any single user that wants to acquire \{p_1, \ldots, p_i\} will have to be authorized (directly or by role hierarchy) for \{r_1, \ldots, r_n\}. ■

By Theorem 4, we can see that canonical SMER constraints of larger cardinality provides additional expressive power over canonical SMER constraints of smaller cardinalities.

5. CONSTRAINT GENERATION

In this section, we give algorithms to generate SMER constraints to enforce SSOD policies.

5.1 SSOD requirements

SSOD policies are expressed in terms of restrictions on permissions. On the other hand, SMER constraints are expressed in term of restrictions on role memberships. In order to generate SMER constraints for enforcing SSOD policies, the first step is to translate restrictions on permissions expressed in SSOD policies to restrictions on role memberships. Such role-level SSOD requirements were introduced in [17].

Definition 10. A k-n SSOD (k-out-of-n Role-based Static Separation of Duty) requirement has the form

\[
\text{ssod}(\{r_1, \ldots, r_n\}, k)
\]

where each \(r_i\) is a role and \(n\) and \(k\) are integers such that \(1 < k \leq n\). The meaning is that there should not exist a set of fewer than \(k\) users that together have memberships in all the \(n\) roles in the requirement. We also say \(k\) users are required to cover the set of \(n\) roles.

We say that an RBAC state \(\gamma\) is safe with respect to the above SSOD requirement when

\[
\forall u_1 \cdots u_{k-1} \in U \left( \bigcup_{i=1}^{k-1} \text{auth}_{\gamma}(u_i) \not\subseteq \{r_1, \ldots, r_n\} \right).
\]

An RBAC state \(\gamma\) is safe with respect to a set \(D\) of SSOD requirements if it is safe with respect to every requirement in \(D\), and we write this as \(\text{safe}_D(\gamma)\).

As role memberships are determined by \(UA\) and \(RH\) only, we sometimes write \(\text{safe}_D(UA, RH)\) as \(\text{safe}_D(UA, RH)\).

Given an SSOD configuration \(\langle PA, RH, E \rangle\), we say that it is equivalent to a set \(D\) of SSOD requirements if

\[
\forall UA \subseteq U \times R \ [ \text{safe}_D((UA, PA, RH)) \Leftrightarrow \text{safe}_D((UA, PA, RH))]\]

where \(\Leftrightarrow\) means logical equivalence.

Similar to Definition 8, we have the following definition:

Definition 11. Let \(RH\) be a role hierarchy, \(D\) be a set of SSOD requirements, and \(C\) be a set of SMER constraints, we say that \(C\) implements \(D\) under \(RH\) when \(C\) is compatible with \(RH\), and

\[
\forall \text{RBAC state } \gamma \ [ \text{safe}_C(\gamma) \Rightarrow \text{safe}_D(\gamma) ]
\]

An algorithm for generating SSOD requirements that are equivalent to SSOD configurations has been given in [18]. We thus focus on generating SMER constraints from SSOD requirements for the rest of this section.

5.2 Comparing SMER Constraints

Given an SSOD configuration \(E, PA, RH\), we first generate a set \(D\) of SSOD requirements, then we need to generate SMER constraints that enforce \(D\) and are compatible with \(RH\). Furthermore, we want to avoid generating constraints that are overly restrictive. If two sets of constraints both implement \(D\) under \(RH\), and one set is less restrictive than the other, then we prefer the less restrictive one. For this, we need to be able to compare two sets of SMER constraints.

Definition 12. Let \(RH\) be a role hierarchy. Let \(C_1\) and \(C_2\) be two sets of SMER constraints. We say that \(C_1\) is at least as restrictive as \(C_2\) under \(RH\) (denoted by \(C_1 \succeq_{RH} C_2\)) if

\[
\forall UA \ [ \text{satisfies}_{C_1}(UA, RH) \Rightarrow \text{satisfies}_{C_2}(UA, RH) ]
\]

The \(\succeq\) relation among all sets of SMER constraints is a partial order. When \(C_1 \succeq_{RH} C_2\) but not \(C_2 \succeq_{RH} C_1\), we say that \(C_1\) is more restrictive than \(C_2\) under \(RH\) (denoted by \(C_1 \gtrsim_{RH} C_2\)).

When neither \(C_1 \gtrsim_{RH} C_2\) nor \(C_2 \gtrsim_{RH} C_1\), we say \(C_1\) and \(C_2\) are incomparable under \(RH\), and we write \(C_1 \not\approx_{RH} C_2\).

In the following, we show how to compare two sets of SMER constraints. It was shown in [17] that for any SMER constraint there exists a set of canonical constraints that is equivalent to it. Therefore, without loss of generality, we compare two sets of canonical SMER constraints. We first show how to compare two individual canonical SMER constraints.

Definition 13. Give a role hierarchy \(RH\), we use \(up_{(RH)}(R)\) to denote the set of all roles that are senior to some role in \(R\), and \(down_{(RH)}(R)\) to denote the set of all roles that are junior to some role in \(R\). More precisely, we define two functions \(up_{(RH)} : 2^R \rightarrow 2^R\) and \(down_{(RH)} : 2^R \rightarrow 2^R\) as follows:

\[
up_{(RH)}(R) = \{ r \mid \exists r' \in R \ [ r \gtrsim_{RH} r' ] \}
\]

\[
down_{(RH)}(R) = \{ r \mid \exists r' \in R \ [ r' \gtrsim_{RH} r ] \}
\]

We omit the subscript \(RH\) when it is obvious from the context.

Lemma 5. For any \(RH\) and canonical SMER constraints \(c_1 = \text{smert}(r_1, k_1)\) and \(c_2 = \text{smert}(r_2, k_2)\), the following hold:

1. \(c_1 \gtrsim_{RH} c_2\) if and only if \(\text{down}(r_1) \subseteq \text{down}(r_2)\).
2. \(c_1 \gtrsim_{RH} c_2\) if and only if \(\text{down}(r_1) \subseteq \text{down}(r_2)\).
3. \(c_1 \equiv_{RH} c_2\) if and only if \(\text{down}(r_1) = \text{down}(r_2)\), which is true if and only if \(r_1 = r_2\).
4. \(c_1 \not\approx_{RH} c_2\) if and only if neither \(\text{down}(r_1) \subseteq \text{down}(r_2)\) nor \(\text{down}(r_1) \supseteq \text{down}(r_2)\).

Proof. We first prove assertion 1. For the “if” direction: Given \(\text{down}(r_1) \subseteq \text{down}(r_2)\), we show that \(\forall UA \ [ \neg \text{satisfies}_{c_2}(UA, RH) \Rightarrow \neg \text{satisfies}_{c_1}(UA, RH) ]\). For any \(UA\), if \(\text{satisfies}_{c_2}(UA, RH)\) is false, then there is a user in \(UA\) who is authorized for all roles in \(r_2\). This user is also authorized for all roles in \(r_1\). Therefore, \(\text{satisfies}_{c_1}(UA, RH)\) is also false.

For the “only if” direction: Suppose, for the sake of contradiction, that \(c_1 \gtrsim_{RH} c_2\) and \(\text{down}(r_1) \not\subseteq \text{down}(r_2)\). It follows that \(r_1 \not\subseteq \text{down}(r_2)\), because if \(r_1 \subseteq \text{down}(r_2)\), then \(\text{down}(r_1) \subseteq \text{down}(\text{down}(r_2)) = \text{down}(r_2)\). Consider a user assignment relation \(UA\) that has a single user \(u\), which is assigned to all roles in \(r_2\). Clearly, \(\text{satisfies}_{c_2}(UA, RH)\) is false. In \((UA, RH)\), the set of all roles that the user \(u\) is authorized for is
down (R₂), satisfies₂₃(UA, RH) is true because the only user in UA is u and u is not authorized for all roles in R₁. This contradicts the assumption that c₁ ≳ RH c₂.

Assertions 2, 3, and 4 follow from Definition 12 and basic facts from set theory. □

Lemma 6. For any RH and two sets of canonical SMER constraints C₁ and C₂, C₁ ≳ RH C₂ if and only if for every c₂ ∈ C₂, there exists c₁ ∈ C₁ such that c₁ ≳ RH c₂.

Proof. The “if” part is clear. For the “only if” part, prove by contradiction. Suppose C₁ ≳ RH C₂ and there exists some c₂ ∈ C₂ and there does not exist c₁ ∈ C₁ such that c₁ ≳ RH c₂. Then we construct a user assignment UA such that there is just one user u. u is assigned with all the roles in c₂. Then UA does not satisfy c₂ since it violates c₂. But UA satisfies every constraint in C₁. Here we get a contradiction, c₁ ≳ RH c₂ does not hold. □

Theorem 7. Given two canonical SMER constraints sets C₁, C₂ and the role hierarchy RH, it takes time O(|C₁| ⋅ |C₂| ⋅ |RH|) to calculate down(RH) to decide if C₁ ≳ RH C₂.

Proof. There are O(|C₁|) SMER constraints in C₁, and there are O(|RH|) roles in RH, thus it takes O(|C₁| ⋅ |RH|) to calculate down(RH)(R₁) and store the result for every c₁ = smer(R₁, k₅) ∈ C₁. Similarly, it takes O(|C₂| ⋅ |RH|) to calculate down(RH)(R₂) and store the result for every c₂ = smer(R₂, k₅) ∈ C₂. To decide if C₁ ≳ RH C₂, it is enough to decide if c₁ ≳ RH c₂ for every c₁ ∈ C₁ and c₂ ∈ C₂. Each comparison takes O(|RH|) and there are at most O(|C₁| ⋅ |C₂|) comparisons. Thus it takes O(|C₁| ⋅ |RH| + |C₂| ⋅ |RH| + |C₁| ⋅ |C₂| ⋅ |RH|) = O(|C₁| ⋅ |C₂| ⋅ |RH|) to decide if C₁ ≳ RH C₂. □

5.3 Normal form of SMER constraints sets

Consider the RH in Figure 1, suppose we have the following SMER constraints:

c₁ = smer(⟨{r₃, r₄}, 2⟩, 2)
c₂ = smer(⟨{r₁, r₂, r₃, r₄}, 4⟩)
c₃ = smer(⟨{r₁, r₃, r₄}, 3⟩, 3)

Although those three constraints look different, in the sense of restrictiveness under RH, they are equivalent. Because r₄ dominates both r₁ and r₂, each of the three constraints says that a single user cannot have all roles of r₁, r₂, r₃, r₄. Suppose c₁, c₂, c₃ are all in a constraints set, two of them are redundant. To remove this redundancy, we can require that for a canonical SMER constraints set smer(⟨{R}, |R|⟩) R = down₃(RH)(R). So we will use c₂ in the constraints set.

Suppose we have another constraint c₄ = smer(⟨{r₂, r₃}, 2⟩, r₂ ≳ RH r₃, c₄. If c₂ and c₄ both appear in a constraints set, c₂ is redundant because c₄ is more restrictive.

To have a simple form of constraints set, we define a normal form for constraints set as following:

Definition 14. A SMER constraints set C is in normal form under RH if and only if

- ∀c ∈ C, c is a canonical SMER constraint.
- ∀c₁, c₂ ∈ C, c₁ and c₂ are not comparable under RH

By the definition, any constraints set C can be converted into normal form. And C and its normal form are equivalent under RH.

5.4 Most restrictive constraints

Given the notion of restrictiveness, given RH and C, we have the most restrictive constraints set as following:

Cₘ = {smer(⟨{R}, |R|⟩ | R ⊆ RH ∧ smer(⟨{R}, |R|⟩) is compatible with RH}

Cₘ is compatible with RH, and it is most restrictive among all compatible constraints sets because given any set C of constraints which is compatible with RH, Cₘ ≳ RH C.

Let Cₘ be the normal form of Cₘ. As discussed in Section 5.3, we would use Cₘ to present the most restrictive constraints set.

For example, in Figure 1, the most restrictive constraints set is {smer(⟨{r₁, r₃}, 2⟩, smer(⟨{r₁, r₃}, 2⟩, smer(⟨{r₂, r₃}, 2⟩, smer(⟨{r₃}, 2⟩)}. In Figure 2, the most restrictive constraints set is {smer(⟨{r₁, r₂, r₃}, 3⟩, Cₘ = {smer(⟨{r₁, r₂}, 2⟩ | r₁, r₂ ∈ R}.

Given the role set RH and a role hierarchy RH, there is a unique minimal most restrictive constraints set Cₘ. A SSOd policies set E can be implemented if and only if Cₘ can implement E.

5.5 Minimal implementation

5.6 Algorithm 1

We now give an algorithm to generate all SMER constraint sets that minimally implement a given SSOd configuration (E, PA, RH), by minimally implementing the set D of SSSOd requirements corresponding to the configuration. The high-level idea of the algorithm is as follows. Given D, RH, the algorithm first computes the most restrictive SMER constraints set. The algorithm then tries to remove or weaken the constraints in the set, until we cannot remove or weaken any constraint (any less restrictive set would not enforce the SSOd policy); this should give a minimal set of constraints that enforces D. By systematically enumerating all ways of doing this, the algorithm generates all such minimal sets of constraints.

We also note that this algorithm can be used when one has a set C of constraints that enforces the desired SSOd configuration but may be too restrictive. One can use the algorithm to compute all constraint sets that are less restrictive than C but still enforce the desired SSOd configuration.

5.7 Algorithm 2

Consider the following scenario. A system administrator already has specified certain constraints; however, these existing con-
By adding the constraint in making $D$ while being unsafe wrt. $C$, there must be some user assignment $UA$ such that $UA$ satisfies $C$ while being unsafe wrt. $D$. Let $u_1$ be a user in $UA$ that is involved in making $D$ unsafe, and let $R_1$ be the set of authorized roles of $u_1$. By adding the constraint $c_1 = \text{ser}(R_1, |R_1|)$, we are able to rule out $UA$. Note that if all roles in $R_1$ share a common ancestor then we cannot add $c_1$, as it is incompatible with $RH$. However, when the configuration is implementable, we can always find a user $u_1$ and prevent $u_1$ from being assigned roles $R_1$.

By repeatedly adding constraints to $C$, $C$ will finally implement $D$ under $RH$, provided that $D$ is implementable. Each time the algorithm will find a number of constraints that can be added to $C$ according to the counter example $UA$. There are two approaches to choose which constraint to add. In the enumeration approach, the algorithm tries all possibilities. In this approach, the algorithm eventually outputs all constraint sets that include the starting constraint set as a subset and minimally implement $D$. In the interaction approach, each time the system will list all possible constraints and let the administrator choose which constraint to add.

6. IMPLEMENTATION OF CONSTRAINT GENERATION

We have implemented the two algorithms in Section 5.6 and Figure 5.7. For the second algorithm, we have implemented both the enumeration variant and the interaction variant. The code is written in C++. Both algorithms need to check whether a set of constraints implements a set of RSSoD requirements. As shown in [17], this problem can be reduced to the propositional satisfaction (SAT) problem. We use the open source SAT solver MiniSAT [7] to solve the SAT problem for this problem. The tool reads an input file that contains the SSoD policies, the permission assignment relation, and the role hierarchy, and outputs all constraint sets that minimally implement the SSoD policies.

Figure 3 and Figure 4 give two example SSoD configurations and all constraint sets that are minimal in implementing the configurations. The example in Figure 3 has an empty role hierarchy relation, and the example in Figure 4 has a non-empty role hierarchy relation.

7. CONCLUSIONS

We have studied a number of problems related to generating SMER constraints for enforcing SSoD policies, while respecting the existing role hierarchy. Particularly we have introduced and implemented two algorithms for generating constraint sets that minimally implement a set of SSoD policies under the given permission assignment and role hierarchy.

Acknowledgement. Portions of this work are supported by NSF CNS-0448204 and sponsors of CERIAS. We thank the anonymous reviewers for their helpful comments.

![Diagram](attachment:image.png)
8. REFERENCES


