Resiliency Policies in Access Control

NINGHUI LI and QIHUA WANG
Purdue University
and
MAHESH TRIPUNITARA
Motorola Labs

We introduce the notion of resiliency policies in the context of access control systems. Such policies require an access control system to be resilient to the absence of users. An example resiliency policy requires that upon removal of any s users, there should still exist d disjoint sets of users such that the users in each set together possess certain permissions of interest. Such a policy ensures that even when emergency situations cause some users to be absent, there still exist independent teams of users that have the permissions necessary for carrying out critical tasks. The Resiliency Checking Problem determines whether an access control state satisfies a given resiliency policy. We show that the general case of the problem and several subcases are intractable (**NP**-hard), and identify two subcases that are solvable in linear time. For the intractable cases, we also identify the complexity class in the polynomial hierarchy to which these problems belong. We discuss the design and evaluation of an algorithm that can efficiently solve instances of nontrivial sizes that belong to the intractable cases of the problem. Furthermore, we study the consistency problem between resiliency policies and static separation of duty policies. Finally, we combine the notions of resiliency and separation of duty to introduce the resilient separation of duty policy, which is useful in situations where both fault-tolerance and fraud-prevention are desired.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Complexity of proof procedures*

General Terms: Security, Theory

Additional Key Words and Phrases: Access control, fault-tolerant, policy design

ACM Reference Format:
Li, N., Wang, Q., and Tripunitara, M. 2009. Resiliency policies in access control. ACM Trans. Inf. Syst. Secur. 12, 4, Article 20 (April 2009), 34 pages. DOI = 10.1145/1513601.1513602.
http://doi.acm.org/10.1145/1513601.1513602.

Author's address: Q. Wang, Purdue University; email: wangq@purdue.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1094-9224/2009/04-ART20 $5.00 DOI: 10.1145/1513601.1513602.
http://doi.acm.org/10.1145/1513601.1513602.

1. INTRODUCTION

While policy analysis has been a main research area in access control for several decades, almost all existing work focuses on properties which ensure that users who should not have access do not get access. For example, safety analysis [Harrison et al. 1976; Lipton and Snyder 1977; Sandhu 1988a] studies whether an access right can be leaked to unauthorized users. Separation of duty (SoD) policies [Clark and Wilson 1987; Saltzer and Schroeder 1975] ensure that no single user (or a set of users of size less than some threshold) is able to perform a sensitive task. Such focus on safety properties probably stems from the fact that access control has been mostly viewed as a tool for restricting access. However, an equally important aspect of access control is to enable access (selectively).

We introduce the notion of resiliency policies which state properties about enabling access in access control. Resiliency policies require that the access control state is resilient to absent users. For example, the access control system of an institution has three separate permissions regarding release of funds: one permission is an endorsement that the request for funds is legitimate, the second permission is the issuance of a check, and the third one is for logging the transaction. The institution's financial office, which takes charge of funding, is composed of a senior treasurer and a number of junior treasurers. In compliance of the separation of duty principle, the senior treasurer has all permissions except the one for logging, while each of the junior treasurers has only one of the three permissions. As issuing funds is a critical task, the institution would like to ensure that even if a few (e.g., two) treasurers (that may include the senior treasurer) are absent (e.g., due to sickness), the remaining personnel in the financial office still have enough privileges to release funds.

Another example resiliency policy requirement is as follows: There must exist three mutually disjoint sets of users such that each set has no more than four users and the users in each set together have all permissions to carry out a critical task. Such a policy would be needed when one needs to be able to send up to three teams of users to different sites to perform a certain task, perhaps in response to some events. One needs to ensure that each team has enough permissions to perform the task, and each team consists of no more than four users (e.g., due to the limit of transportation means).

Such policies are particularly useful when evaluating whether the access control configuration of a system is ready for emergency response. When an emergency such as a natural diaster or a terrorist attack occurs, an organization may need to send out teams of employees to respond to the emergency. At the same time, such an emergency may prevent employees from reporting to work, causing them to be absent. These policies ensure that even when emergency situations cause some users to be absent, there still exist independent teams of users that have the necessary permissions for carrying out critical tasks. In other words, these policies mandate that there is a certain level of redundancy in assigning permissions to users so that the system can tolerate some users being absent.

Some level of resilience to absent users must already exist in most enterprise authorization management system. For example, every organization needs to make sure that when certain key personnel is absent, the organization can still function. However, to the best of our knowledge, such resiliency requirements have not been formalized in the computer security literature before. Our contributions in this article are as follows:

(1) We introduce the notion of Resiliency Policies which express requirements about enabling access rather than restricting access. We give a concrete formulation for a resiliency policy which captures the intuition discussed above.
(2) We study computational complexities of the Resiliency Checking Problem, which determines whether an access control state satisfies a given resiliency policy. We show that this problem is **NP**-hard in the general case and is in $\mathbf{coNP}^{\mathbf{NP}}$, a complexity class in the Polynomial Hierarchy. We show that several subcases are **NP**-complete. We identify two subcases that are solvable in linear time.
(3) We show that, notwithstanding the intractability results, many instances of the Resiliency Checking Problem of nontrivial sizes may still be efficiently solvable. We present an algorithm for the Resiliency Checking Problem. Our algorithm uses a pruning technique that reduces the number of combinations that need to be considered. The experimental results show that this pruning technique can reduce the search space by several orders of magnitude. Our algorithm also takes advantage of the observation that the problem of checking whether the state can tolerate the removal of a particular absent set can be naturally formulated as the boolean satisfiability problem. This enables us to use existing SAT solvers in our implementation and benefit from several decades of research in designing SAT solvers. Our experimental results show that our algorithm can efficiently solve instances of nontrivial sizes.
(4) Resiliency policies may conflict with safety-oriented policies such as static separation of duty (SSoD) policies [Li et al. 2004]. We study the policy consistency problem between resiliency policies and SSoD policies and we demonstrate how to simplify the problem and present criteria for determining consistency for a number of special cases. Finally, we show that determining consistency is both **NP**-hard and **coNP**-hard, and is in $\mathbf{NP}^{\mathbf{NP}}$.
(5) In many situations, both fault-tolerance and fraud-prevention are required. Even though one can express the two requirements using a resiliency policy plus a separation of duty policy, it is more desirable to use a single policy to capture the requirements. We introduce the resilient separation of duty policy to serve the purpose. This new type of policy is the combination of a separation of duty policy and a particular form of resiliency policy. We study the satisfaction problem as well as the optimization problem on resiliency separation of duty policies.

The remainder of this article is organized as follows. In Section 2, we define resiliency policies and the Resiliency Checking problem. We present computational complexities of the Resiliency Checking problem in Section 3 and an algorithm for the problem and an implementation of the algorithm in Section 4. In Section 5, we explore the policy consistency problem. In Section 6, we introduce the resilient separation of duty policy. We discuss related work in Section 7. Finally, we conclude and present open problems related to the concept of resiliency in Section 9.

2. RESILIENCY POLICIES AND THE RESILIENCY CHECKING PROBLEM

Definition 1 (Resiliency Policies). A resiliency policy takes the form

$$\mathsf{rp}\left\langle P, s, d, t \right\rangle$$

where rp is a keyword, $P = \{p_1, \ldots, p_n\}$ is a set of permissions, $s \geq 0$ and $d \geq 1$ are integers, and t is either a positive integer or the special symbol ∞.

We say that an access control state satisfies such a resiliency policy if and only if upon removal of any set of s users, there still exist d mutually disjoint sets of users such that each set contains no more than t users and the users in each set together are authorized for all permissions in P.

Intuitively, a resiliency policy $\mathsf{rp}\langle P, s, d, t\rangle$ specifies a fault tolerance requirement with respect to a certain critical task. The set P includes all permissions that are needed to carry out the task. The faults that we would like to tolerate are absent users. The parameter s specifies the number of absent users that we want to be able to tolerate. The parameter d is motivated by the requirement that several teams may be needed to carry out multiple instances of the task. If only one team is needed, then d can be set to 1. The parameter t specifies the size limit of each team. This is motivated by limitations on the maximal number of users that can be involved in any instance of task. If no such limitation exists, then t can be set to ∞.

We initially formed the notion of resiliency policies (as given in Definition 1) in the wake of Hurricane Katrina. What kind of resiliency requirements should one place on the authorization system to respond to such emergency? First, some number of users may be absent, motivating the parameter s. Second, the organization may need to send out multiple teams to respond to the emergency, hence the parameter d. Each team must possess all permissions in P to be able to effectively respond, and the team size may be limited by transportation or other cost considerations, hence the parameter t. Resiliency requirements are certainly also needed in nonemergency situations, for which we given an example below.

Example 1 (Business Office). Consider the access control state of a business office from Figure 1. It relates to the example we introduce in Section 1. To issue funds, all three permissions *Endorse*, *Issue*, and *Log* must be possessed by a set of users. In our resiliency policy, we set $P = \{Endorse, Issue, Log\}$. If we set $s = 1$ in our policy, then we want the system to be resilient to the absence of any (one) user. If we set $d = 2$, this means that we require two sets

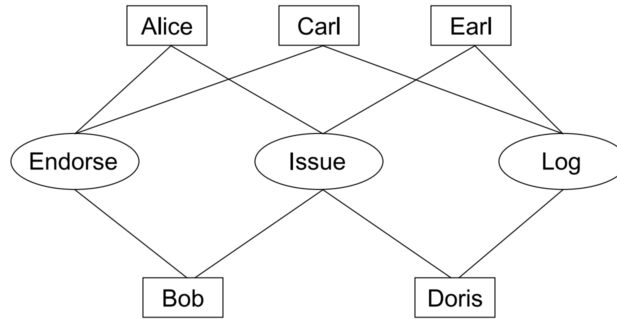

Fig. 1. An example of an access control state with five users, *Alice*, *Bob*, *Carl*, *Doris* and *Earl*, and 3 permissions, *Endorse*, *Issue* and *Log*. A line segment connects a user (e.g., *Alice*) to a permission (e.g., *Endorse*) to indicate that the user has the permission. This corresponds to the example from Section 1 on releasing funds; all three permissions must be possessed by a group of users that together want to release funds.

of users such that users in each set together possess all permissions. If we set $t = \infty$, this means that the set of users that together possess all permissions can be of any size.

We observe that in our example, $\mathsf{rp}\langle P, 1, 2, \infty\rangle$ is satisfied. For instance, after removing *Alice*, the two users *Carl* and *Earl* together have all three permissions, as are *Bob* and *Doris*. The cases in which another user is removed can be verified similarly. However, $\mathsf{rp}\langle P, 2, 2, \infty\rangle$ is not satisfied because if *Alice* and *Bob* are absent, the only user that possesses *Endorse* is *Carl*, and one user cannot belong to two disjoint sets. Similarly, $\mathsf{rp}\langle P, 2, 1, \infty\rangle$ is satisfied, but $\mathsf{rp}\langle P, 3, 1, \infty\rangle$ is not satisfied because if *Alice*, *Bob*, and *Carl* are absent, then no user possesses *Endorse*. And finally, we observe that $\mathsf{rp}\langle P, 1, 1, 2\rangle$ is satisfied, but not $\mathsf{rp}\langle P, 1, 1, 1\rangle$ because for the latter case, there exists no single user that has all three permissions.

The two parameters s and d are related. If an access control state satisfies $\mathsf{rp}\langle P, s, d, t\rangle$, then it also satisfies $\mathsf{rp}\langle P, s+i, d-i, t\rangle$ for any i such that $0 < i < d$. For example, if, after removing any two users, there exist three mutually disjoint sets of users such that each set covers all permissions in P, then after removing any three users, there are at least two sets left. However, if a state satisfies $\mathsf{rp}\langle P, s+1, d-1, t\rangle$, it may not satisfy $\mathsf{rp}\langle P, s, d, t\rangle$. For our example shown in Figure 1, we observe that $\mathsf{rp}\langle P, 1, 2, \infty\rangle$ is satisfied. However $\mathsf{rp}\langle P, 0, 3, \infty\rangle$ is not satisfied because we need the three users *Alice*, *Bob*, and *Carl* that possess *Endorse* to belong to distinct sets; this still leaves one permission that needs to be covered by each set, and we have only two users that remain.

Resiliency policies can be defined in any access control system in which there are users and permissions. This includes almost all access control systems, including Discretionary Access Control systems [Lampson 1971; Graham and Denning 1972] and Role-Based Access Control systems [Sandhu et al. 1996]. We assume that an access control state is given by a binary relation $UP \subseteq \mathcal{U} \times \mathcal{P}$, where $\mathcal{U}$ represents the set of all users, and $\mathcal{P}$ represents the set of all

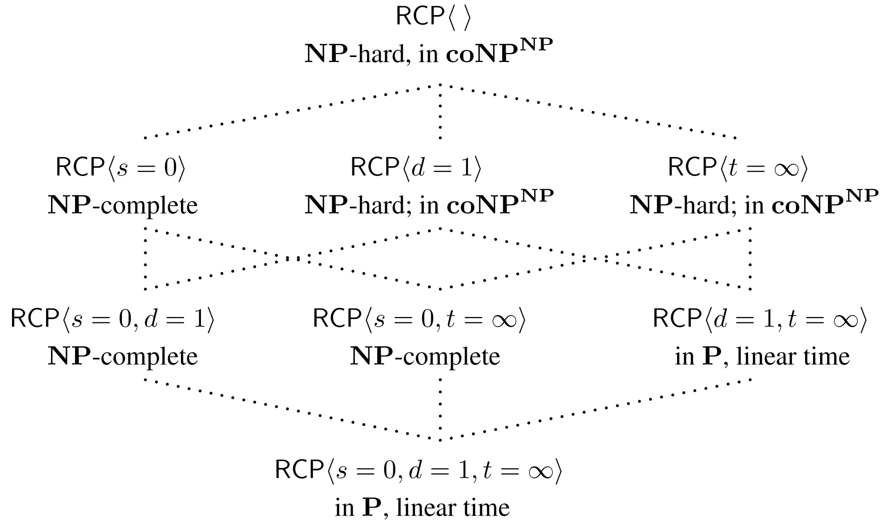

Fig. 2. Time complexity of the Resiliency Checking Problem (RCP) and its various subcases.

permissions. Note that by assuming that a state is given by a binary relation $UP \subseteq \mathcal{U} \times \mathcal{P}$, we are not assuming permissions are directly assigned to users; rather, we assume only that one can calculate the relation *UP* from the access control state.

Definition 2 (Resiliency Checking Problem (RCP)). Given a resiliency policy r and an access control state *UP*, determining whether *UP* satisfies r is called the Resiliency Checking Problem (RCP).

A resiliency policy has three parameters: s, d, and t. In some situations, one may need to consider only those policies with one or more of these parameters degenerated. The parameter s, which denotes the number of absent users that the system needs to tolerate, may be degenerated to always be 0. The parameter d, which denotes the number of sets of users required, may be degenerated to always be 1. Finally, the parameter t, which denotes the size bound on each set, may be degenerated to always be ∞. There are eight cases where some of the three parameters are degenerated. For example, a resiliency policy in the subcase RCP$\langle s = 0, d = 1\rangle$ has the form rp$(P, 0, 1, t)$, which asks whether there exists a set of users of size at most t that together have all permissions in P; while the subcase RCP$\langle t = \infty\rangle$ asks whether there exists several distinct sets of users (d sets) each of whose users together have all permissions in P, even after any set of s users is removed from the state. In particular, RCP$\langle\ \rangle$ is the general case of the problem.

3. COMPUTATIONAL COMPLEXITIES OF THE RESILIENCY CHECKING PROBLEM

The following theorem summarizes the computational complexity results for RCP and its various subcases. These results are also shown in Figure 2.

THEOREM 1. *The computational complexities of the Resiliency Policy Checking problem are as follows.*

—$\mathsf{RCP}\langle\ \rangle$, *the most general case, is* **NP***-hard and is in* $\mathbf{coNP^{NP}}$*, as are the two subcases* $\mathsf{RCP}\langle d = 1\rangle$ *and* $\mathsf{RCP}\langle t = \infty\rangle$.
—$\mathsf{RCP}\langle s = 0, d = 1\rangle$, $\mathsf{RCP}\langle s = 0, t = \infty\rangle$, *and* $\mathsf{RCP}\langle s = 0\rangle$ *are* **NP***-complete.*
—$\mathsf{RCP}\langle d = 1, t = \infty\rangle$ *and* $\mathsf{RCP}\langle s = 0, d = 1, t = \infty\rangle$ *can be solved in linear time.*

Our complexity results show that RCP is in $\mathbf{coNP^{NP}}$. This means that the complement of RCP can be solved by a nondeterministic Oracle Turing Machine that has oracle access to a machine that can answer any **NP** queries. (See Appendix A for a brief overview of Oracle Turing Machines.) Intuitively, given an access control state and a resiliency policy $r = \mathsf{rp}(P, s, d, t)$, to decide nondeterministically that the state does not satisfy r, one can guess a set of s users to be removed, and then query the **NP** oracle whether the remaining users contain d mutually disjoint sets of users such that each set is of size at most t and the users in each set together have all the permissions in P.

Another way to understand the computational complexity of RCP is to observe that an RCP instance has the form $\forall$ size-s subset, $\exists d$ sets of users that satisfy some requirements that can be efficiently verified. Problems in **NP** have the form of $\exists$ an evidence that satisfies some polynomial-time verifiable requirements. Problems in **coNP** has the form $\forall$ choices, some polynomial-time verifiable requirements hold. RCP has one alternation of $\forall$ followed by $\exists$, which makes it in $\mathbf{coNP^{NP}}$.

We have shown that RCP (and its two subcases $\mathsf{RCP}\langle d = 1\rangle$ and $\mathsf{RCP}\langle t = \infty\rangle$) are **NP**-hard and are in $\mathbf{coNP^{NP}}$. It remains open whether these three problems are $\mathbf{coNP^{NP}}$-complete or not. Readers who are familiar with computational complexity theory will recognize that $\mathbf{coNP^{NP}}$ is a complexity class in the Polynomial Hierarchy. (See Appendix 9 for a brief introduction to the Polynomial Hierarchy.) Because the Polynomial Hierarchy collapses when $\mathbf{P} = \mathbf{NP}$, showing that an **NP**-hard decision problem is in the Polynomial Hierarchy, although is not equivalent to showing that the problem is **NP**-complete, has the same consequence: the problem can be solved in polynomial time if and only if $\mathbf{P} = \mathbf{NP}$.

In the rest of this section, we prove the results in Theorem 1. The following lemmas prove that $\mathsf{RCP}\langle s = 0\rangle$ is in **NP**, $\mathsf{RCP}\langle s = 0, d = 1\rangle$ and $\mathsf{RCP}\langle s = 0, t = \infty\rangle$ are **NP**-hard, $\mathsf{RCP}\langle\ \rangle$ is in $\mathbf{coNP^{NP}}$, and $\mathsf{RCP}\langle d = 1, t = \infty\rangle$ is in **P**. The complexities of other subcases can be implied from these results.

LEMMA 2. $\mathsf{RCP}\langle s = 0\rangle$ *is in* **NP**.

PROOF. An instance consists of an access control state UP and a policy $\mathsf{rp}\langle P, 0, d, t\rangle$. UP satisfies $\mathsf{rp}\langle P, 0, d, t\rangle$ if and only if there exist d mutually disjoint sets of users such that the users in each set together cover all permissions in P and each set has at most t users. If these d sets are given, they can be verified in polynomial time. Therefore, $\mathsf{RCP}\langle s = 0\rangle$ is in **NP**. □

LEMMA 3. $\mathsf{RCP}\langle s = 0, d = 1\rangle$ *is* **NP***-hard.*

PROOF. We reduce the **NP**-complete set covering problem [Papadimitriou 1994] (also referred to as minimum covering problem in Garey and Johnson [1979]) to $\mathsf{RCP}\langle s = 0, d = 1\rangle$. In set covering, we are given a set S, n subsets of S: $S_1, \ldots, S_n$, and a budget K, and need to determine whether the union of K subsets is the same as S. An instance of $\mathsf{RCP}\langle s = 0, d = 1\rangle$ asks whether an access control state UP satisfies a policy $\mathsf{rp}\langle P, 0, 1, t\rangle$. In our reduction, each element in S is mapped to a permission in P and each subset S_i is mapped to a user u_i. In other words, if the subset S_i contains an element, then u_i is authorized for the permission corresponding to the element. We now argue that the mapping ensures that there exists a set of users of size at most K together have all the permissions in P if and only if K subsets cover S. Assume that a set of users of size at most K exists such that those users together have all the permissions in P. Then, we pick the subsets that are mapped to those users, and their union gives us S. For the other direction, assume that K subsets cover S. Then, the K users to which the subsets are mapped together have all the permissions in P. □

LEMMA 4. $\mathsf{RCP}\langle s = 0, t = \infty\rangle$ *is* **NP**-*hard.*

PROOF. We reduce the **NP**-complete domatic number problem [Garey and Johnson 1979] to $\mathsf{RCP}\langle s = 0, t = \infty\rangle$. Given a graph $G(V, E)$, the domatic number problem asks whether V can be partitioned into k disjoint sets $V_1, V_2, \cdots, V_k$, such that each V_i is a dominating set for G. V' is a dominating set for $G = (V, E)$ if for every node u in $V - V'$, there is a node v in V' such that $(u, v) \in E$. An instance of $\mathsf{RCP}\langle s = 0, t = \infty\rangle$ asks whether an access control state UP satisfies a policy $\mathsf{rp}\langle P, 0, d, \infty\rangle$. Given a graph $G = (V, E)$, we construct an access control state UP with n users $u_1, u_2, \cdots, u_n$ and n permissions $p_1, p_2, \cdots, p_n$, where n is the number of nodes in V. Each user corresponds to a node in G, and $v(u_i)$ denotes the node corresponding to user u_i. In UP, user u_i is authorized for the permission p_j if and only if either $i = j$ or $(v(u_i), v(u_j)) \in E$. Let P denote the set $\{p_1, p_2, \cdots, p_n\}$. A dominating set in G corresponds to a set of users that together have all the permissions in P. UP satisfies $\mathsf{rp}\langle P, 0, k, \infty\rangle$ if and only if V contains k disjoint dominating sets. □

LEMMA 5. $\mathsf{RCP}\langle\,\rangle$ *is in* $\mathbf{coNP^{NP}}$.

PROOF. We show that the complement of $\mathsf{RCP}\langle\,\rangle$ is in $\mathbf{NP^{NP}}$. Assume that we have an oracle that decides the Resiliency Checking problem when $s = 0$, which, as we know, is **NP**-complete. We construct a nondeterministic oracle Turing machine M that accepts UP and $\mathsf{rp}\langle P, s, d, t\rangle$ when UP does not satisfy $\mathsf{rp}\langle P, s, d, t\rangle$. M nondeterministically removes s users, and then queries the oracle. If the oracle machine returns "yes", M rejects; otherwise, M accepts, because it has found a set of users, the removal of which violates the Resiliency policy. The construction of M shows that the complement of $\mathsf{RCP}\langle\,\rangle$ is in $\mathbf{NP^{NP}}$. Therefore, $\mathsf{RCP}\langle\,\rangle$ is in $\mathbf{coNP^{NP}}$. □

LEMMA 6. $\mathsf{RCP}\langle d = 1, t = \infty\rangle$ *can be solved in linear time.*

To prove this lemma, we introduce the notion of tolerance bounds. Intuitively, a tolerance bound is the number of users that are authorized for the permission that is assigned to the fewest users.

Definition 3 (The Tolerance Bound). Given an access control state UP and a set $\{p_1, \cdots, p_m\}$ of permissions, we define the *tolerance bound* of UP and $\{p_1, \cdots, p_m\}$, denoted by $tb(UP, \{p_1, \cdots, p_m\})$, to be $\min_{1 \le i \le m} \#(p_i)$, where $\#(p_i)$ denotes the number of users who are authorized for p_i in the state UP.

Given an $\mathsf{RCP}\langle d = 1, t = \infty\rangle$ instance that asks whether UP satisfies $\mathsf{rp}\langle P, s, 1, \infty\rangle$, the answer is yes if and only if the tolerance bound is greater than s. If the tolerance bound is at most s, then there is a permission that is assigned to at most s users, removing all these users results in no user having that permission. On the other hand, if the tolerance bound is greater than s, then after removing any set of s users, each permission is still assigned to at least one user, which means that the set of all remaining users together have all the permissions in P.

The tolerance bound can be computed in linear time. A straightforward algorithm is to first go through all pairs in UP to count how many users each permission is assigned to, maintaining a counter for each permission, and then return the minimal value among the counters. This, together with the above observations, suffices to prove Lemma 6.

The tolerance bound can be used to answer other RCP instances negatively as well. Given an RCP instance that asks whether UP satisfies $\mathsf{rp}\langle P, s, d, t\rangle$, if $s + d > tb(UP, P)$, then the answer is "no", as removing s users can result in fewer than d users have a particular permission. On the other hand, when $d \ge 2$ and $s + d \le tb(UP, P)$, we do not immediately know whether UP satisfies $\mathsf{rp}\langle P, s, d, t\rangle$ or not.

3.1 Implications of the Complexity Results

As can be seen in Figure 2, RCP is tractable when only one instance of a task must be performed ($d = 1$) and the team-size is not a concern ($t = \infty$). The RCP problem becomes intractable when either t or d is not degenerated and is an input to the problem.

Note that the team-size limitation is needed only when one wants to limit the number of involved users to be less than $|P|$, the number of permissions needed for the task. To see this, every user in a team should has a permission in P that is not authorized to any other members in the team, or the user may be removed from the team without affecting the team's capability to complete the task. Also note that the parameter d is needed only when the teams of users for performing difference instances of a given task must be disjoint. We believe that in many practical cases, the resiliency requirements are likely to not require explicit team size limit or disjoint teams, in which case such resiliency policies fall into the tractable side.

However, even if a resiliency requirement does not fall into the tractable case, that is, it needs disjoint teams or team-size limitation, it does not mean that such a requirement cannot be encoded and checked in practice. That RCP

is intractable (**NP**-hard) in general does not mean that all instances are infeasible to solve in practice. The intractability result simply means that there exist difficult problem instances that take exponential time in the worst case. Many instances that will be encountered in practice may still be efficiently solvable. Experiences have shown that many important problems, such as boolean satisfaction (SAT) and integer linear programming (ILP), which are intractable in theory, have algorithms that are efficient in practice. We will introduce an algorithm for RCP and report its performance in the next section.

4. AN ALGORITHM FOR RCP

We now describe an algorithm for RCP. We first describe an algorithm for the subcase $\mathsf{RCP}\langle t = \infty\rangle$, that is, there is no limit on the number of users in any of the d mutually disjoint sets. We then describe how to extend the algorithm to deal with the parameter t when it is not degenerated. In Section 4.2 we discuss our implementation of this algorithm and its effectiveness using experimental results.

4.1 Description of the Algorithm

To determine whether UP satisfies $\mathsf{rp}\langle P, s, d, \infty\rangle$, a straightforward algorithm is to enumerate all sets of s users, and for each such set A (which we call an *absent set*), remove the users in A from UP and check whether among the remaining users there are d mutually disjoint sets of users such that each set covers the permissions. If the answer is "no" for any absent set, then we know that UP does not satisfy $\mathsf{rp}\langle P, s, d, \infty\rangle$. If we have enumerated through all absent sets, and the answer is "yes" for each of them, then we know that UP satisfies $\mathsf{rp}\langle P, s, d, \infty\rangle$.

Our algorithm, which is based on the above straightforward approach, has a number of improvements which greatly reduce the runtime. First, we preprocess the given access control state to remove irrelevant users and permissions. Second, we use a static pruning technique so as to minimize the number of absent sets that need to be considered. Finally, for each absent set A, we reduce the problem of checking whether the given access control state can tolerate the removal of A to an instance of the propositional satisfiability (SAT) problem. SAT is the problem of determining if the variables of a given propositional formula can be assigned in such a way that makes the formula evaluate to TRUE. In the rest of this section, we discuss our improvements in detail.

4.1.1 *Preprocessing*. Given the state UP and the policy $\mathsf{rp}\langle P, s, d, \infty\rangle$, we first remove (u, p) from UP if $p \notin P$, as we do not need to consider permissions not in the policy. Also, we only consider those users who are authorized for at least one permission in P. Finally, we calculate the tolerance bound $tb(UP, P)$, using the methods described in the end of Section 3. If $s + d > tb(UP, P)$, then we know the answer is "no".

4.1.2 *Reduction to SAT*. A key step to solve RCP is to determine whether, after removing a certain set of users, there still exist d mutually disjoint sets of users such that each set covers all permissions in P. We observe that such

a problem can be translated into a SAT instance. This enables us to benefit from the extensive research on SAT and to use existing SAT solvers. SAT has been studied extensively for several decades (see, for example, Du et al. [1997]), and many clever algorithms have been developed. Problems in many fields, including databases, planning, computer-aided design, machine vision, and automated reasoning, have been reduced to SAT and solved using SAT solvers. Often times, this results in better performance than using existing domain-specific algorithms for those problems.

The translation works as follows. Let U be the set of users after removing users in an absent set. For each user u_i in U and each integer j from 1 to d, we have a propositional variable $v_{i,j}$. This variable is true if the i'th user is assigned to the j'th group. Then we have the following two kinds of clauses. The first kind of clauses ensures that all permissions are covered in each of the d groups: For each permission p in P, let $u_{i_1}, u_{i_2}, \cdots, u_{i_x}$ be users in U who are authorized for p. Then for each j from 1 to d, we add the clause $v_{i_1,j} \vee v_{i_2,j} \vee \cdots \vee v_{i_x,j}$. There are $|P| \cdot d$ of such clauses. The second kind of clauses ensure that no user is selected in two groups at the same time: For each user u_i, and for each pair k, ℓ such that $0 < k < \ell \leq d$, we add the clause $\neg v_{i,k} \vee \neg v_{i,\ell}$. There are $nd(d-1)/2$ such clauses, where n is the number of users. It is clear that the total number of clauses added is polynomial to the size of the RCP instance.

4.1.3 *Static pruning.* The number of size-s user sets among n users is close to n^s when s is small compared with n. For example, there are more than one billion such sets for $s = 6$ and $n = 100$. We observe that not all these sets need to considered. There is a partial order relation among these sets such that if A_1 *dominates* A_2, and the RCP instance can tolerate the removal of A_1, then it can also tolerate the removal of A_2. This means that we only need to consider A_1. We now explain this pruning technique.

Definition 4 (Absent Set Domination). Among all users in UP, we say a user u_1 *dominates* another user u_2 if u_1's set of permissions is a superset (not necessarily strict superset) of u_2's. We say a set of users, A_1, *dominates* another set A_2 if there is a bijection between users in A_2 and A_1 such that for every user u in A_2, the corresponding user in A_1 dominates the user u.

LEMMA 7. *Assuming that A_1 dominates A_2, if an* RCP *instance can tolerate removing A_1, then it can also tolerate removing A_2.*

PROOF. We need to show that, if after removing A_1, there are d mutually disjoint sets of users such that each set covers all permissions in P, then after removing A_2, there are also d mutually disjoint sets each of which covers all permissions in P.

By definition, if A_1 dominates A_2, then there exists a bijection f between A_2 and A_1, such that $f(u) = v$ implies user $v \in A_1$ dominates user $u \in A_2$. Without loss of generality, we assume that f satisfies the property that if $u \in A_1 \cap A_2$, then $f(u) = u$. Observe that if f does not satisfy this property for some $u \in A_1 \cap A_2$, then there exist $u_1 \in A_1$ and $u_2 \in A_2$ such that $f(u) = u_1$

and $f(u_2) = u$. It follows that u_1 dominates u and u dominates u_2. Because the domination relation is transitive, we have u_1 dominating u_2. We can then assign $f(u) = u$ and $f(u_2) = u_1$. By repeating this process, we can arrive at a bijection f such that if $u \in A_1 \cap A_2$, then $f(u) = u$. This property implies that if $u \in A_2 \backslash A_1$, then $f(u) \in A_1 \backslash A_2$.

Let $S_1, \cdots, S_d$ be the disjoint sets of users after the removal of A_1, we now construct $S_1', \cdots, S_d'$ such that (1) these sets consists only of users not in A_2, (2) they are mutually disjoint, and (3) users in each set together have all the permissions in P.

For each $k \in [1, d]$, S_k' is constructed as follows: for every user u in S_k, if $u \in A_2$, then u is replaced with $f(u)$. Observe that because $u \in S_k$, then $u \notin A_1$, and thus $u \in A_2 \backslash A_1$ and $f(u) \in A_1 \backslash A_2$. Therefore, each S_k' includes only users not in A_2. To show that they are mutually disjoint, we need to show, for each $w \in S_k'$, that w does not appear in S_j', where $j \neq k$. There are two cases. Case 1: w is the result of replacing $x \in A_2$, in which case $w = f(x)$ is a member of A_1, implying w does not appear in S_j. Hence, if w also appears in S_j', it must also be from replacement of x. This is impossible, because x cannot appear both in S_k and S_j. Case 2: w appears in S_k, in which case $w \notin S_j$. Furthermore, $w \notin A_1$, and therefore w cannot be used as replacement for any other user. Therefore, w does not appear in S_j'. Finally, by definition of dominance, user $f(u)$'s set of permissions is a superset of u's. Since S_k has all permissions in P, S_k' also has all permissions in P. □

4.1.4 *Enumerate all absent sets that need to be considered.* We would like to systematically generate only size-s user sets that we need to consider. That is, we need to ensure that (1) any size-s user set is dominated by at least one generated user set, and (2) we do not generate two sets such that one of them dominates the other. The naïve way of finding all such sets is to generate all size-s user sets and, for each such set, check whether it is dominated by any other size-s set. However, this would be very inefficient. We now describe an algorithm that directly generates only the user sets that need to be considered.

The algorithm works as follows. First of all, we sort all users based on the number of permissions they have, in decreasing order, and assign each user an index, that is, users are listed as $u_0, \cdots, u_{n-1}$. If $0 \leq i < j \leq n-1$, then u_i has at least as many permissions as u_j. By definition of dominance, if u_i dominates u_j, then either $i < j$ or u_i and u_j have exactly the same set of permissions. Second, we use an index e that initially has the value $s-1$. We generate the first size-s set $\{u_0, \cdots, u_e\}$, and then increase the index e by one each time and generate all user sets that include u_e and are not dominated by any other set generated before. A key observation is that we only need to generate user sets that have the *closure property*. We now explain this observation.

Definition 5 (Closure Property). Given a set of users $U = \{u_0, \cdots, u_{n-1}\}$, we say a set $A \subseteq U$ has the *closure property* if and only if for any $u_k \in A$, and any $u_i \in U$ such that $i < k$ and u_i dominates u_k, we have $u_i \in A$.

In other words, if a set A has the closure property, then any user that dominates a user in A and comes before that user must also be in A. The

relationships between the closure property and the set dominance relation are established in the following two lemmas.

LEMMA 8. *Let A be a size-s user set that satisfies the closure property and let e be the index of the user with largest index in A, then there is no size-s subset of* $\{u_0, u_1, \cdots, u_{e-1}\}$ *that dominates A.*

PROOF. Because A satisfies the closure property, then u_e and all users among $\{u_0, u_1, \cdots, u_{e-1}\}$ that dominate u_e are also in A. Let k be the number of users in $\{u_0, u_1, \cdots, u_{e-1}\}$ that dominate u_e, then A has $k+1$ users that dominate u_e (including u_e itself). By the definition of set domination, any set that dominates A must have at least $k+1$ users that dominate u_e. Whereas any subset of $\{u_0, u_1, \cdots, u_{e-1}\}$ has at most k users that dominate u_e. Therefore, no subset of $\{u_0, u_1, \cdots, u_{e-1}\}$ dominates A. □

Lemma 8 shows that if A satisfies the closure property, then none of the sets that have been considered so far dominates A, so A needs to be considered.

LEMMA 9. *Let A be a size-s user set that does not satisfy the closure property and let e be the index of the user with largest index in A, then there exists a size-s subset of* $\{u_0, u_1, \cdots, u_{e-1}, u_e\}$ *that dominates A and satisfies the closure property.*

PROOF. Since A does not have the closure property, there is a user $u_k \in A$ such that there exists u_i such that $i < k$, u_i dominates u_k, and $u_i \notin A$. We change A to A_1 by substituting u_k with u_i, that is, $A_1 = A \backslash \{u_k\} \cup \{u_i\}$. Clearly, A_1 dominates A. If A_1 still does not satisfy the closure property, we can repeat the substitution process until the resulting set has closure property. □

Lemma 9 shows that if A does not satisfy the closure property, then there must exist a set that dominates A and either has been considered or will be generated and considered, so there is no need to consider A. The above two lemmas together show that we need to generate only the users sets that satisfy the closure property.

4.1.5 *Dynamic pruning.* When an absent set A is generated, we invoke a SAT solver to evaluate whether after users in A are removed, the remaining users still satisfy the requirements. If the answer is "yes", then we would get back a solution, which consists of d sets of users such that each set covers all permissions. Let E be the set of all users that appear in any of the d sets; we call E a solution set for A. Let U be the set of all users in UP. Clearly, $E \subseteq U - A$. If E contains fewer users than $U - A$, then it is possible that when another set A' is generated we have $E \cap A' = \emptyset$. When this happens, we know that we do not need to consider A', as E is also a solution set for A'. Based on this observation, one can store the solution sets returned by the SAT solver, and use them to check whether absent sets generated later need to be considered.

4.1.6 *Handling the case that $t \neq \infty$.* The reduction to SAT described above works only when $t = \infty$. To handle the case that $t \neq \infty$, we can use

pseudo-boolean constraints. In Pseudo-Boolean (PB) constraints, all variables take values of either 0 (false) or 1 (true). Constraints are linear inequalities with integer coefficients, for example, $2x + y + z \geq 2$ is a PB constraint. A disjunctive clause encountered in SAT is a special case of PB constraints; for example, $x \vee y \vee z$ is equivalent to $x + y + z \geq 1$. Many SAT solvers also support PB constraints. In particular, the SAT solver we use, SAT4J [Le Berre 2006], supports PB constraints.

When $t \neq \infty$, we can translate the problem of determining whether d sets of size no more than t exist to the satisfiability problem with PB constraints. The translation works as follows. For each user u_i and each integer j from 1 to d, we have a propositional variable $v_{i,j}$. This variable is true if the i'th user is assigned to the j'th group. Then we have the following three kinds of constraints. The first kind ensures that all permissions are covered: For each permission p in P, let $u_{i_1}, u_{i_2}, \cdots, u_{i_x}$ be the users who are authorized for the permission p. Then, for each j from 1 to d, we add the constraint $v_{i_1,j} + \cdots + v_{i_x,j} \geq 1$. There are $|P| \cdot d$ of such constraints. The second kind ensures that each set contains at most t users: for each j from 1 to d, we add the constraint $v_{0,j} + v_{1,j} + \cdots + v_{n-1,j} \leq t$. There are d such constraints. The third kind ensures that no user is selected in two groups: For each user i, add the constraint $v_{i,1} + \cdots + v_{i,d} \leq 1$. There are n such constraints, where n is the number of users.

4.2 Implementation and Evaluation

We have implemented the algorithm described in Section 4.1, and performed several experiments using randomly generated instances. Our goals of implementing the algorithm and performing these experiments are to understand the effectiveness of the pruning techniques developed in Section 4 and to understand how well the algorithm scales with different parameters.

The implementation of our algorithm was written in Java. We use SAT4J [Le Berre 2006], an open source satisfiability library in Java. Experiments were carried out on a PC with an Intel Pentium 4 CPU running at 3.2 GHz with 1 GB of RAM running Microsoft Windows XP Professional 2002. Our time units are milliseconds. In this subsection, n, s, and d denote the number of total users, the number of users that may be absent, and the number of disjoint sets of users we seek after the removal of a set of users respectively. The methodology that we use in generating testing instances is explained in Appendix B.

Our experimental results show that our algorithm is able to solve nontrivial size of RCP instance in reasonable amount of time. For example, our implementation spent approximately 500ms on instances with 60 to 100 users, 10 permissions, $s = 3$ and $d = 6$; and around 2 seconds on instances with 80 to 100 users, 10 permissions, $s = 3$ and $d = 4$. We discuss our observations from the experiments in the rest of this section.

4.2.1 *The algorithm scales reasonably well with n when d is small; however when d is more than approximately 8, the algorithm stops scaling.* The runtime of the algorithm depends on the total number of absent sets that need

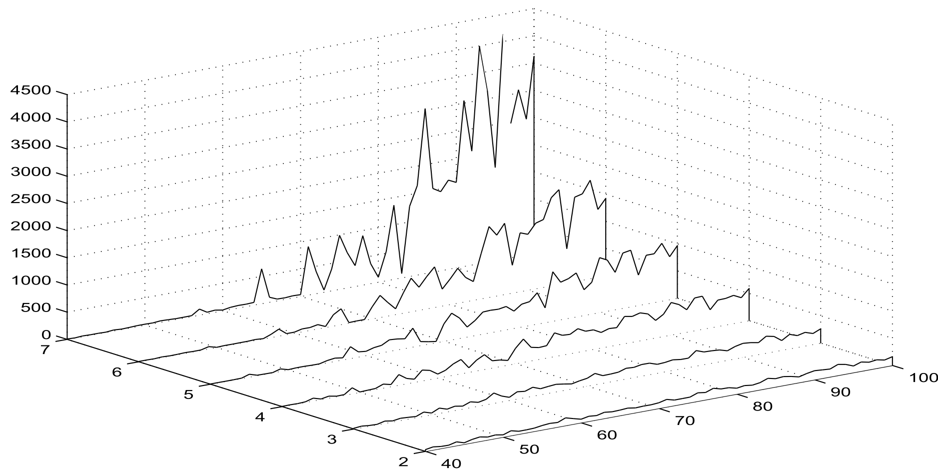

Fig. 3. This graph shows the effect on runtime (in milliseconds) as the number of users n and the number of disjoint sets d increase. The size of absent sets is 3 and there are 10 permissions. The value of n increases from 40 to 100 and the value of d increases from 2 to 7.

to be examined and the time it takes for the SAT solver to solve each SAT instance. The time spent in the SAT solver is greatly influences by d, which is the number of distinct sets of users we seek after an absent set of users is removed. In Figure 3, we plot the runtime of the algorithm for cases in which the instance is true, for increasing n (number of users) and d. For smaller values of n (say, n = 40), increasing d has almost no effect on the runtime so long as d is no larger than 7. The reason is that relatively few absent sets need to be considered. However, for larger values of n (say, n = 90), increasing d has a pronounced effect on the runtime. In particular, we observe that up to a particular value for d (7 in this case), the algorithm scales well as n increases. For example, for n = 100 and d = 6, the algorithm takes only about 1.7 seconds. However, as d becomes larger, the algorithm stops scaling. A major reason is that as d increases beyond a certain threshold (8 in our case), each SAT instance that is generated is time-consuming for the SAT solver to solve. Consequently, lots of time is spent in the SAT solver, which results in increase of runtime of our algorithm. This threshold of around 8 seems to hold for many other experiments we have performed.

We expect d to be small in most practical cases, as people normally do not need a large number of disjoint groups to perform different instances of a task. Since our algorithm scales well when d is small, it should perform well in practice.

4.2.2 *Static pruning is very effective.* Table I shows the effect of static pruning for increasing values of n (number of users) and s (size of absent sets). While static pruning always reduces the number of absent sets to be considered, its effect is especially pronounced for large values of n and s. For example, for n = 100 and s = 8, we see a reduction of 7 orders of magnitude in the number of absent sets that need to be considered. We point out also that the effect

Table I. A Table that Shows that Static Pruning is Effective

s \ n	40	60	80	100
2	45	28	40	36
	780	1770	3160	4950
4	1042	694	684	640
	$9.1 \cdot 10^4$	$4.9 \cdot 10^5$	$1.6 \cdot 10^8$	$3.9 \cdot 10^6$
6	9713	9248	5310	6653
	$3.8 \cdot 10^6$	$5.0 \cdot 10^7$	$3.0 \cdot 10^8$	$1.2 \cdot 10^9$
8	$7.7 \cdot 10^6$	$6.1 \cdot 10^4$	$1.2 \cdot 10^5$	$8.7 \cdot 10^4$
	$7.7 \cdot 10^7$	$2.6 \cdot 10^9$	$2.9 \cdot 10^{10}$	$1.9 \cdot 10^{11}$

Note: The columns are values for n (number of users) and rows are values for s (size of the absent set). The number of permissions is 10. For each cell in the table, the entry above the dotted line is the number of absent sets that need to be considered with static pruning in effect, and the number below the dotted line is the number of absent sets to be considered without pruning (i.e., $\binom{n}{s}$).

of static pruning is increasingly pronounced for larger values of n when s is constant. For example, for $s = 6$ and increasing n from 40 to 100, the reduction in the number of absent sets that need to be considered improves from a difference of 3 orders of magnitude to 6. For a fixed number of permissions (10 in this case), occurrences of dominance may increase as n increases (because there are likely more users who have a lot of permissions that dominate other users). This explains why the number of absent sets after pruning is fewer, for example, for $s = 4, n = 100$ (640 absent sets) than for $s = 4, n = 40$ (1,042 absent sets).

4.2.3 *Dynamic pruning is not effective.* The basic idea of dynamic pruning is to store, for each absent set A, the set E of users that are used in the solution returned by the SAT solver. When encountering another absent A', we check whether $A' \cap E = \emptyset$; if so, then we can skip A'. Somewhat unexpected for us, it turns out that dynamic pruning is not effective. In fact, using dynamic pruning is often slower than without dynamic pruning. After analyzing this effect, the reason became clear. Dynamic pruning adds additional processing time for each absent set. It is cost effective only when invoking the SAT solver is expensive so that it is worthwhile to take more effort to further decrease the number of absent sets needed to be examined. However, when invoking the SAT solver is expensive, that is, when it is difficult to find d mutually disjoint sets of users such that each set has all permissions, the solution returned by the SAT solver likely includes all users that are not in A, which means that this solution set will not be able to prune any other absent set.

5. ON THE CONSISTENCY OF RESILIENCY AND SEPARATION OF DUTY POLICIES

As we have discussed in the introduction, resiliency policies are a natural complement to traditional safety policies in access control. Consequently, a question arises regarding the consistency of resiliency policies with other

policies. In this section, we explore the consistency of resiliency policies and static separation of duty (SSoD) policies.

The intent of an SSoD policy is to preclude any group of users from possessing too many permissions. We adopt the concrete formulation of such policies from Li et al. [2004]. An SSoD policy is of the form $\mathsf{ssod}\langle P, k\rangle$, where P is a set of permissions and $1 < k \leq |P|$ is an integer. An access control state satisfies the policy if there exists no set of fewer than k users that together possess all permissions in P. In the policy $\mathsf{ssod}\langle P, k\rangle$, P denotes the set of permissions that are needed to perform a sensitive task, and k denotes the minimal number of users that are allowed to perform the task. If the policy is satisfied, then no set of $k-1$ users can together perform the task, because they do not have all the permissions; thus at least k users need to be involved, achieving the goal of separation of duty. For example, the policy $\mathsf{ssod}\langle\{p_1, p_2\}, 2\rangle$ means that no single user is allowed to have both p_1 and p_2.

In many cases, it is desirable for an access control system to have both resiliency and SSoD policies. If an access control system has only resiliency policies, then they can be satisfied by giving all permissions to all users, resulting in each single user able to perform any task. Similarly, if an access control system has only SSoD policies, then they can be satisfied by not giving any permission to any user, resulting in no task able to be performed. It is clear that neither kind of policy by itself is sufficient to capture the security requirements. When both kinds of policies coexist, safety and functionality requirements can all be specified.

Due to their opposite focus, resiliency policies and separation of duty policies can conflict with each other. For example, a separation of duty policy $\mathsf{ssod}\langle P, 2\rangle$ requires that no user possess all permissions in P. A resiliency policy $\mathsf{rp}\langle P, s, d, 1\rangle$ requires the existence of a user that has all permissions in P. Clearly, the two policies cannot be satisfied simultaneously. We formally define our notion of consistency amongst such policies in the following definition.

Definition 6. Given a set F of resiliency and separation of duty policies, the policies in F are consistent if and only if there exists an access control state UP such that UP satisfies every policy in F. Determining whether F is consistent is called the *Policy Consistency Checking Problem* (PCCP).

The following lemma asserts that the actual value of s and d in a resiliency does not affect its compatibility with SSoD policies. This enables us to replace all resiliency policies in the form of $\mathsf{rp}\langle P_i, s_i, d_i, t_i\rangle$ in F with the special form $\mathsf{rp}\langle P_i, 0, 1, t_i\rangle$ when studying PCCP. This greatly simplifies the problem.

LEMMA 10. *F is a set of policies and $R = \mathsf{rp}\langle P, s, d, t\rangle \in F$. Let $R' = \mathsf{rp}\langle P, 0, 1, t\rangle$ and $F' = (F - \{R\}) \cup \{R'\}$. F is consistent if and only if F' is consistent.*

PROOF. It is clear that if F is consistent then F' is consistent. In the following, we prove that if F' is consistent then F is consistent. Assume that state UP' satisfies all policies in F'. UP' satisfying R' implies that there is a set U of no more than t users together have all the permissions in P. We then construct a new state UP by adding $s + d - 1$ copies of all users in U to UP'. Note that

adding copies of existing users in UP' will not lead to violation of SSoD policies in F'. In this case, UP satisfies R plus all policies in F'. In other words, UP satisfies all policies in F and F is consistent. □

The following theorem gives the computational complexity results about general cases of PCCP. Observe that the case with one SSoD policy and an arbitrary number of resiliency policies is **coNP**-hard, and the case with one resiliency policy and an arbitrary number of SSoD policies is **NP**-hard. Therefore, it is unlikely that the general case is in **NP** or in **coNP**; however, we show that the problem is in $\mathbf{NP}^{\mathbf{NP}}$.

THEOREM 11. *The computational complexities for* PCCP *are as follows:*

(1) PCCP $\langle 1, n \rangle$ *is* **coNP**-*hard, where* PCCP $\langle 1, n \rangle$ *denotes the subcase that there is a single SSoD policy, and an arbitrary number of resiliency policies.*
(2) PCCP $\langle m, 1 \rangle$ *is* **NP**-*hard, where* PCCP $\langle m, 1 \rangle$ *denotes the subcase that there is an arbitrary number of SSoD policies, and a single resiliency policy.*
(3) PCCP $\langle m, n \rangle$*, that is, the most general case of* PCCP*, is in* $\mathbf{NP}^{\mathbf{NP}}$.

We prove Theorem 11 by proving Lemmas 12, 13, and 14. Without loss of generality, we assume that for any static separation of duty policy $\mathsf{sod}\langle P, k\rangle$, we have $k \leq |P|$. We also assume that in any resiliency policy $\mathsf{rp}\langle P, s, d, t\rangle$, we have either $t = \infty$ or $t \leq |P|$.

LEMMA 12. PCCP $\langle 1, n \rangle$ *is* **coNP**-*hard, where* PCCP $\langle 1, n \rangle$ *denotes the subcase that there is a single SSoD policy, and an arbitrary number of resiliency policies.*

PROOF. We reduce the **NP**-complete set covering problem [Papadimitriou 1994] (also referred to as minimum covering problem in Garey and Johnson [1979]) to the complement of PCCP. In set covering, we are given a set $X = \{e_1, \cdots, e_m\}$, n subsets of X: $X_1, \ldots, X_n$, and a budget b, and need to determine whether the union of b subsets is the same as X. Given an instance of the set covering problem, we construct one SSoD policy $S = \mathsf{sod}\langle P, b+1\rangle$ and b rp policies $R_i = rp\langle P_i, 0, 1, 1\rangle$ $(1 \leq i \leq b)$, where $P = \{p_1, \cdots, p_m\}$ corresponds to X and $P_i = \{p_j \mid e_j \in X_i\}$ corresponds to X_i. Let $F = \{S, R_1, \cdots, R_n\}$. In the following, we prove that F is inconsistent if and only if the answer to the set covering problem is "yes".

On the one hand, if F is inconsistent, there does not exist any state that satisfies all polices in F. In other words, if a state satisfies all resiliency policies in F, there exists no more than b users in the state who together have all the permission in P. Let UP be a state with n users $u_1, \cdots, u_n$ such that $(u_i, p_j) \in UP$ if and only if $p_j \in P_i$. It is clear that UP satisfies all resiliency policies in F, and hence there exist no more than b users together have all the permissions in P. In other words, there exist no more than b elements in $\{P_1, \cdots, P_n\}$ whose union is P. Thus, the answer to the set covering problem is "yes".

On the other hand, if the answer to the set covering problem is "yes", then there exist no more than b elements in $\{P_1, \cdots, P_n\}$ whose union is P. For any state UP that satisfies all resiliency policies in F, let U be the set of users that

satisfy at least one resiliency policy. $u \in U$ if and only if there exists P_i such that u has all permissions in P_i. In this case, there exist no more than b users in U who together have all the permissions in P. Hence, UP does not satisfy S, which implies that no state satisfies all policies in F. □

LEMMA 13. PCCP $\langle m, 1 \rangle$ *is* **NP**-*hard, where* PCCP $\langle m, 1 \rangle$ *denotes the subcase that there is an arbitrary number of SSoD policies, and a single resiliency policy.*

PROOF. We reduce the **NP**-complete set splitting problem to PCCP. In the set splitting problem, we are given a set $X = \{e_1, \cdots, e_n\}$, m subsets of X: $X_1, \ldots, X_m$, and need to determine whether there exist Y_1 and Y_2 such that $Y_1 \cup Y_2 = X$ and there does not exist X_i $(1 \leq i \leq m)$ such that $X_i \subseteq Y_1$ or $X_i \subseteq Y_2$. Given an instance of the set splitting problem, construct a resiliency policy $R = \mathsf{rp}\langle P, 0, 1, 2\rangle$ and m SSoD policies $S_i = \mathsf{sod}\langle P_i, 2\rangle$ $(1 \leq i \leq m)$, where $P = \{p_1, \cdots, p_n\}$ corresponds to X and $P_i = \{p_j \mid e_j \in X_i\}$ corresponds to X_i. Let $F = \{R, S_1, \cdots, S_m\}$. In the following, we prove that F is consistent if and only if the answer to the set splitting problem is "yes".

On the one hand, if F is consistent, then there exists a state UP that satisfies all policies in F. UP satisfying R implies that there exist two users u_1 and u_2 in UP such that u_1 and u_2 together have all the permissions in P. Furthermore, UP satisfying S_i implies that neither u_1 nor u_2 has all permissions in P_i. Let $Y_1 = \{e_i \mid (u_1, p_i) \in UP\}$ and $Y_2 = \{e_i \mid (u_2, p_i) \in UP\}$. We have $Y_1 \cup Y_2 = X$ and neither Y_1 nor Y_2 is a superset of any X_i. The answer to the set splitting problem is "yes".

On the other hand, if the answer to the set splitting problem is "yes", then such Y_1 and Y_2 exist. We construct a state UP containing only two users u_1 and u_2 such that $(u_i, p_j) \in UP$ $(1 \leq i \leq 2)$ if and only if $p_j \in Y_i$. Since $Y_1 \cup Y_2 = X$, u_1 and u_2 together have all the permissions in P. Furthermore, since there does not exist X_i such that X_i is a subset of Y_1 or Y_2, neither u_1 nor u_2 has all permissions in P_i, which implies that UP satisfies S_i. Therefore, UP satisfies all policies in F. □

LEMMA 14. *Let* $F = \{S_1, S_2, \cdots S_m, R_1, \cdots, R_n\}$, *where* $S_i = \mathsf{sod}\langle P_i, k_i\rangle$ $(1 \leq i \leq m)$ *and* $R_j = \mathsf{rp}\langle Q_j, s_j, d_j, t_j \ (1 \leq j \leq n)\rangle$. *Checking whether policies in* F *are consistent is in* $\mathbf{NP}^{\mathbf{NP}}$.

PROOF. We construct a set of policies F' by replacing every R_i $(1 \leq i \leq n)$ in F with $\mathsf{rp}\langle P_i, 0, 1, t_i\rangle$. From Lemma 10, F is consistent if and only if F' is consistent.

We construct a nondeterministic Oracle Turing machine M that makes use of an **NP** oracle machine to determine whether F' is consistent. M first nondeterministically selects an integer a such that $max(k_1, \cdots, k_m) \leq a \leq \Sigma_{i=1}^{n}|Q_i|$ and then generates a users. Note that at least $max(k_1, \cdots, k_m)$ users are needed to satisfy all SSoD policies in F', and at most $\Sigma_{i=1}^{n}|Q_i|$ users are needed to satisfy all resiliency policies in F'. (The state can have more than $\Sigma_{i=1}^{n}|Q_i|$ users, but in order to show that all resiliency policies in F' are satisfied, at most $\Sigma_{i=1}^{n}|Q_i|$ users need to be involved.) Then M constructs a state UP by

nondeterministically assigning a subset of Q to u, where $Q = \bigcup_{i=1}^{n} Q_i$ is the set of all permissions that appear in the resiliency policies. Next, M nondeterministically construct n sets $U_1, \cdots, U_n$ of users in UP, and then, for every $i \in [1, n]$, checks whether users in U_i together have all the permissions in P_i and $|U_i| \leq t_i$. If the answer is "no", then M returns `False`. Finally, M invokes the **NP** oracle to check whether UP violates any SSoD policy. (In order to prove that a state violates a static separation of duty policy $\mathsf{sod}\langle P, k\rangle$, we just need to present a set of no more than k users in the state who together have all the permissions in P. Therefore, checking whether a state violates an SSoD policy is in **NP**.) If the oracle machine answers "yes", M returns `False`. Otherwise, M returns `True`, which means that UP satisfies all policies in F' and hence F' is consistent. It is clear that M terminates in polynomial time if the oracle machine returns an answer instantaneously. Therefore, PCCP is in $\mathbf{NP}^{\mathbf{NP}}$ in general. □

6. THE RESILIENT SEPARATION OF DUTY POLICY

In the previous section, we discuss the policy consistency problem in which different policies may apply to different sets of permissions. A set of permissions are usually those that are necessary for a certain task to be accomplished. We call the consistency problem in the previous section the *system-level consistency problem*, as it considers policies that apply to different tasks in the system. In this section, we consider the policy consistency problem at *task-level*, which means that the policies we consider apply to the same task (or, equivalently, the same set of permissions). In particular, we consider the consistency between one static separation of duty policy (SSoD) and one resiliency policy. The combination of an SSoD policy and a resiliency policy is useful in situations where a task requires both fraud-prevention and fault-tolerance. We propose the notion of *resilient separation of duty (ReSoD) policy* to express such a policy combination.

Example 2. Consider the business office scenario in Example 1 where the task of issuing funds requires a set P of three permissions and we have five users. A resiliency policy states that upon the absence of any single user, the remaining users should still have enough permissions to complete the task. Also, in order to prevent frauds, it is required that the task cannot be completed by a single user. These two requirements are captured by a resiliency policy $\mathsf{rp}\langle P, 1, 1, \infty\rangle$ and a static separation of duty policy $\mathsf{ssod}\langle P, 2\rangle$. And it can be verified that the access control state presented in Figure 1 satisfies both policies.

The combination of an SSoD policy $\mathsf{ssod}\langle P, k\rangle$ and a resiliency policy in the form of $\mathsf{rp}\langle P, s, 1, \infty\rangle$ provides both fraud-prevention and fault-tolerance for the task that requires the permissions in P. When both policies are about the same set of permissions, they are closely related to each other, as increasing resiliency may come at a cost of decreasing separation of duty. We thus use a single policy, which we call the ReSoD policy, to express the combination of the two kinds of policies and we also study the property of ReSoD policies, such as

whether they can be satisfied at all. In most of today's enterprise authorization management systems, concerns about SoD and resiliency are often part of the considerations when managing permissions; however, typically they are not formally specified as such specification is not supported by today's authorization management systems. The notion of ReSoD policy enables the formal specification and verification of such requirements.

Definition 7 (ReSoD). A *resilient separation of duty policy (ReSoD)* is given as a tuple $\mathsf{resod}\langle P, k, s\rangle$, where P is a set of permissions, and k and s are integers where $k > 1$ and $s \geq 0$.

An access control state *UP satisfies* $\mathsf{resod}\langle P, k, s\rangle$, if and only if *UP* satisfies both $\mathsf{ssod}\langle P, k\rangle$ and $\mathsf{rp}\langle P, s, 1, \infty\rangle$.

In Example 2, the resiliency and SoD requirements can be expressed as $\mathsf{resod}\langle P, 2, 1\rangle$.

We would like to point out that it is possible to define a more general form of ReSoD policies by also incorporating the two additional parameters d and t that are part of resiliency policies. In such a case, *UP* satisfies $\mathsf{resod}\langle P, k, s, d, t\rangle$ if and only if it satisfies both $\mathsf{ssod}\langle P, k\rangle$ and $\mathsf{rp}\langle P, s, d, t\rangle$. Definition 7 keeps ReSoD policy in a simpler form by setting $d = 1$ and $t = \infty$. Such a simple form is sufficient in many practical situations. Studying the more general form of ReSoD policies is future work.

Given an ReSoD policy, a natural problem that arises is to check whether an access control state *UP* satisfies the policy, which is called the *ReSoD satisfaction problem*. The following theorem states the computational complexity of this problem.

THEOREM 15. *Given an access control state UP and an ReSoD policy* $\mathsf{resod}\langle P, k, s\rangle$*, the problem of checking whether UP satisfies* $\mathsf{resod}\langle P, k, s\rangle$ *is* **coNP**-*complete.*

PROOF. Checking whether *UP* satisfies $\mathsf{ssod}\langle P, k\rangle$ is **coNP**-complete [Li et al. 2004]. Also, from Theorem 1, checking whether *UP* satisfies $\mathsf{rp}\langle P, s, 1, \infty\rangle$ is in **P**. Therefore, the problem is **coNP**-complete. □

Besides the satisfaction problem, another fundamental problem related to ReSoD is the *ReSoD Satisfiability Problem (*ReSAT*)*, which asks whether there exists a state *UP* that satisfies a given ReSoD policy. Clearly, $\mathsf{resod}\langle P, k, s\rangle$ is satisfiable, if and only if $\mathsf{ssod}\langle P, k\rangle$ and $\mathsf{rp}\langle P, s, 1, \infty\rangle$ are consistent (see Definition 6 in the previous section) with each other. The following lemma states a necessary and sufficient condition for satisfiability of ReSoD policies.

LEMMA 16. $\mathsf{resod}\langle P, k, s\rangle$ *is satisfiable if and only if* $|P| \geq k$.

PROOF. Assume for the purpose of contradiction that *UP* satisfies $\mathsf{resod}\langle P, k, s\rangle$, where $|P| < k$. In this case, *UP* contains a set of users who together have all the permissions in P. In other words, every permission in P is assigned to at least one user in *UP*. Therefore, there exists a set of no more than $|P|$ users who together have all the permissions in P. Since $|P| < k$, $\mathsf{ssod}\langle P, k\rangle$ is violated, which indicates that *UP* does not satisfy $\mathsf{resod}\langle P, k, s\rangle$ which is the desired contradiction.

On the other hand, when $|P| \geq k$, we can create a state UP with $|P|(s+1)$ users, such that every permission in P is assigned to exactly $s+1$ users and every user has exactly one permission. In this case, at least $|P|$ users are required to have all the permissions in P. Also, after removing s users, for every permission $p \in P$, there is at least one remaining user that has p. In general, UP satisfies $\mathsf{resod}\langle P, k, s\rangle$. □

Lemma 16 asserts that given $\mathsf{resod}\langle P, k, s\rangle$, as long as the number of permissions in P is at least the minimal number of users required for the task (for concern of separation of duty), then it is always satisfiable no matter how high is the required number of absent users to be tolerated. The proof of this lemma exploits the fact that one can use an unbounded number of users where each user is assigned a single permission. This result is not very interesting as in any practical system the number of users is bounded and one cannot always assign only a single permission to a user. Also observe that any state that has too few (e.g., s) users cannot satisfy $\mathsf{resod}\langle P, k, s\rangle$.

We say that $\mathsf{resod}\langle P, k, s\rangle$ is satisfiable with m users if and only if there exists at least one state with m users that satisfies $\mathsf{resod}\langle P, k, s\rangle$. Clearly, if $\mathsf{resod}\langle P, k, s\rangle$ is satisfiable with m users, it is also satisfiable with $n > m$ users. Let m_o be the smallest m such that $\mathsf{resod}\langle P, k, s\rangle$ is satisfiable with m users. Such a bound m_o must exist, and for any integer $n < m_o$, there does not exist any state with n users that satisfies $\mathsf{resod}\langle P, k, s\rangle$.

This bound m_o can be useful in practice. Consider Example 2 again. Assume that we would like to minimize the number of employees in the business office. Then we need to calculate this bound. Note that three users are sufficient. For example, we may assign *Endorse* to *Alice* and *Bob*, *Issue* to *Alice* and *Carl*, and *Log* to *Bob* and *Carl*. But we cannot construct a state with fewer than three users that satisfies the ReSoD policy. Therefore, the bound is three.

Given $\mathsf{resod}\langle P, k, s\rangle$, calculating the exact bound m_o can be difficult. Whether this is tractable or not is an open problem. In the rest of this section, we present techniques for computing upper-bound m_u and lower-bound m_l such that if $m \geq m_u$, then $\mathsf{resod}\langle P, k, s\rangle$ is satisfiable with m users, and if $m < m_l$, then $\mathsf{resod}\langle P, k, s\rangle$ is not satisfiable with m users. The following lemma computes m_l.

LEMMA 17. $\mathsf{resod}\langle P, k, s\rangle$ *is satisfiable with m users only if* $m \geq k+s$ *and* $m \geq \lceil \frac{(s+1)n}{n-(k-1)} \rceil$, *where* $n = |P|$.

PROOF. When $m < k+s$, removing a set of s users from a set of m users results in a set of fewer than k users. And if these remaining users together have all the permissions in P, then $\mathsf{ssod}\langle P, k\rangle$ is violated.

When $m < \lceil \frac{(s+1)n}{n-(k-1)} \rceil$, as each permission is assigned to $s+1$ users, the total number of assignments is $(s+1)n$. $m < \lceil \frac{(s+1)n}{n-(k-1)} \rceil$ indicates that there exist at least one user u' who has more than $n-(k-1)$ permissions. Let P' be the set of permissions of u'. Since every permission in P has been assigned to users, there exist a set U_1 of no more than $|P-P'| < n-(n-(k-1)) = k-1$ users who together have all the permissions in $P-P'$. In this case, users in $U_1 \cup \{u'\}$

together have all the permissions in P, and $|U_1 \cup \{u'\}| < k$, which violates $\mathsf{ssod}\langle P, k\rangle$. □

Note that when $n < k + s$, we have $\lceil \frac{(s+1)n}{n-(k-1)} \rceil < k + s$, while when $n \geq k + s$, we have $\lceil \frac{(s+1)n}{n-(k-1)} \rceil \geq k + s$. Hence, which of the two lower-bounds is tighter depends on how the value of n (which is $|P|$) compares to $k + s$.

Next, we consider m-ReSAT in special cases and then compute m_u in general cases. First of all, we show that m-ReSAT is essentially a matrix marking problem.

Definition 8 (MMP). Given an $m \times n$ matrix M and two integer k and x, the *Matrix Marking Problem*, denoted as $\mathsf{MMP}\langle m, n, k, x\rangle$, asks whether one can mark the cells in M with either 0 or 1 in such a way that both of the following conditions are true.

—For every column, there are at least x cells marked with 1.

—There does not exist a set R of fewer than k rows, such that for every column, there is a row in R whose cell in that column is marked with 1.

Let $M(i, j)$ denote the cell on the ith row and jth column of matrix M. Intuitively, in Definition 8, the matrix M represents an access control state UP. Rows and columns in M correspond to users and permissions in UP, respectively. $M(i, j)$ being marked with 1 indicates that user u_i has permission p_j in UP. M satisfying the first condition in Definition 8 indicates that UP satisfies $\mathsf{rp}\langle P, s, 1, \infty\rangle$, while M satisfying the second condition indicates that UP satisfies $\mathsf{ssod}\langle P, k\rangle$. The following lemma states that $\mathsf{MMP}\langle m, |P|, k, s + 1\rangle$ is equivalent to m-ReSAT of $\mathsf{resod}\langle P, k, s\rangle$.

LEMMA 18. *Given* $\mathsf{resod}\langle P, k, s\rangle$ *and an integer m, the answer to* m-ReSAT *is "yes", if and only if the answer to* $\mathsf{MMP}\langle m, n, k, s + 1\rangle$ *is "yes", where* $n = |P|$.

The proof for this lemma is straightforward and we omit it in this article .

As the two problems are equivalent, in the rest of this section, we discuss MMP instead of m-ReSAT. Given $\mathsf{MMP}\langle m, n, k, s + 1\rangle$, m_o denotes the smallest m that makes $\mathsf{MMP}\langle m, n, k, s + 1\rangle$ satisfiable. We summarize our results on computing m_o in below. Results in Lemma 17 are included as well.

Exact Answer

—When $s = 0$, $m_o = k$

—When $k = 2$, $m_o = \lceil \frac{(s+1)n}{n-1} \rceil$

—When $k = n$, $m_o = (s + 1)k$

—When $n \geq C^{k+s}_{s+1}$, $m_o = k + s$

Upper-Bound

—When $k \leq n < C^{k+s}_{s+1}$, see Theorem 23 for m_u

Lower-Bound

—When $n \geq k + s$, $m_l = k + s$
—When $k \leq n < k + s$, $m_l = \lceil \frac{(s+1)n}{n-(k-1)} \rceil$

In the rest of this section, we prove the above results.

LEMMA 19. *The following are true.*

—*When $s = 0$,* MMP$\langle m, n, k, s+1\rangle$ *is satisfiable if and only if $m \geq k$.*
—*When $k = 2$,* MMP$\langle m, n, k, s+1\rangle$ *is satisfiable if and only if $m \geq \lceil \frac{(s+1)n}{n-1} \rceil$.*
—*When $k = n$,* MMP$\langle m, n, k, s+1\rangle$ *is satisfiable if and only if $m \geq (s+1)k$.*

PROOF. When $s = 0$ and $m \geq k$, for every $i \in [1, k]$, we mark $M(i, i)$ with 1. For every other column, we arbitrarily mark one cell in the column with 1. In this case, at least k rows are required to have a cell marked with 1 in every column. Therefore, MMP$\langle m, n, k, s+1\rangle$ is satisfiable. Also, from Lemma 17, MMP$\langle m, n, k, s+1\rangle$ is not satisfiable if $m < k + s = k$. Therefore, MMP$\langle m, n, k, s+1\rangle$ is satisfiable if and only $m \geq k$.

When $k = 2$, each row can have as many as $n - 1$ cells marked with 1. Also, the total number of cells marked with 1 in the matrix is $(s + 1)n$. If each row has $n - 1$ cells with 1, $\lceil \frac{(s+1)n}{n-1} \rceil$ rows is enough. Therefore, MMP$\langle m, n, 2, s+1\rangle$ is satisfiable when $m \geq \lceil \frac{(s+1)n}{n-1} \rceil$. However, if $m < \lceil \frac{(s+1)n}{n-1} \rceil$, in order to have $(s + 1)n$ cells marked with 1, there must exist a row having all the n cells marked with 1, which violates the second condition in Definition 8.

When $k = n$, each row can have at most one cell marked with 1. In this case, at least $(s+1)k$ rows are needed to ensure that every column contains $s+1$ cells marked with 1. □

The following lemma states a special case where the number of columns in the matrix is no smaller than a certain threshold.

LEMMA 20. *When $n \geq C^{k+s}_{s+1}$,* MMP$\langle m, n, k, s+1\rangle$ *is satisfiable if and only if $m \geq k + s$, where C^{k+s}_{s+1} is the total number of $(s + 1)$-combinations from a set of $k + s$ elements*[1].

Since $k + s$ is a lower-bound of m when MMP$\langle m, n, k, s+1\rangle$ is satisfiable, we only need to prove the "if" part of Lemma 20. We design an algorithm that marks an $m \times n$ matrix M. The algorithm is described in Figure 4. Generally speaking, the algorithm marks each column of M based on $(s+1)$-combinations from $\{1, \cdots, s+k\}$. The proof for the correctness of the algorithm with respect to MMP is in Appendix 9. The following example illustrates how the algorithm works.

Example 3. Let M be a 4×6 matrix, $k = 3$ and $s = 1$. We have $C^{k+s}_{s+1} = C^4_2 = 6$. The set of 2-combinations out of $\{1, 2, 3, 4\}$ is $E^4_2 = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$. The algorithm $Mark(M, k, s)$ marks the

[1]Given integers x and y $(x \geq y)$, $C^x_y = \frac{x!}{y!(x-y)!}$.

Input: An $m \times n$ matrix M with all cells having value 0, integers k and s, where $n \geq C_{s+1}^{k+s}$ and $m \geq k + s$.

Output: A marked version of M such that for every column, there are $s + 1$ cells marked with 1; and there does not exist a set R of fewer than k rows, such that for every column, there exists a row in R whose cell in that column is marked with 1.

```
Procedure Mark(M, k, s)
    i = 1;
    For every e ∈ E^{k+s}_{s+1} Do
        For j = 1 To k + s Do
            If j ∈ e Then
                Mark M(j, i) with 1;
        i + +;
    EndFor;
    If i < n Then            //at this moment, i = C^{k+s}_{s+1}
        For i = C^{k+s}_{s+1} + 1 To n Do
            Randomly mark s + 1 cells in Column i with 1;
End;
```

Fig. 4. An algorithm that marks an $m \times n$ matrix. $M(i, j)$ denotes the cell in the ith row and jth column of M, and E_y^x is the set of y-combinations from $\{1, \cdots, x\}$.

6 columns of M based on the six elements in E_2^4. For instance, for the first column, $(1, 2)$ indicates marking the cells in row 1 and row 2 with 1. The output of the algorithm is shown in Table II. It can be verified that at least three rows are needed to have a cell marked with 1 in every column.

Next, we use Lemma 20 to compute m_u for MMP$\langle m, n, k, s+1\rangle$ in general cases. The following lemma states an important fact that we can make use of.

LEMMA 21. *Let k_1 and k_2 be integers such that $k_1 > 0, k_2 > 0$ and $k_1+k_2 = k$. We have $C_{s+1}^{k_1+s} + C_{s+1}^{k_2+s} \leq C_{s+1}^{k+s}$.*

Lemma 21 can be easily verified by arithmetic computation. We omit the proof in this article.

According to Lemma 21, even if $n < C_{s+1}^{k+s}$, there may exist $k_1, \cdots, k_x$ such that $k = \Sigma_{i=1}^{x} k_i$ and $n \geq \Sigma_{i=1}^{x} C_{s+1}^{k_i+s}$. This leads to the following Lemma 22, which gives an upper-bound of m_o in general case. Intuitively, we may come up with x matrices $M_1, \ldots, M_x$ such that M_i ($i \in [1, x]$) is an $m_i \times n_i$ matrix that satisfies MMP$\langle m_i, n_i, k_i, s+1\rangle$. We may then place these x matrices inside a bigger matrix M so that none of the x matrices share a row or a column in M. Then, M satisfies MMP$\langle \Sigma_{i=1}^{x} m_i, \Sigma_{i=1}^{x} n_i, \Sigma_{i=1}^{x} k_i, s+1\rangle$.

LEMMA 22. *Let $\{k_1, \ldots, k_x\}$ be a set of integers such that $k_i > 0$ and $k = \Sigma_{i=1}^{x} k_i$. If $n \geq \Sigma_{i=1}^{x} C_{s+1}^{k_i+s}$, then MMP$\langle m, n, k, s+1\rangle$ is satisfiable if $m \geq k + xs$.*

PROOF. Let $\{M_1, \ldots, M_x\}$ be a set of matrices, such that M_i is an $m_i \times n_i$ matrix, where $\Sigma_{j=1}^{x} m_j = m$, $n_i \geq C_{s+1}^{k_i+s}$ and $\Sigma_{j=1}^{x} n_j = n$.

From Lemma 20, since $n_i \geq C_{s+1}^{k_i+s}$, MMP$\langle m_i, n_i, k_i, s\rangle$ is satisfiable if $m_i \geq k_i + s$. Let M_i' be the marked version of M_i such that M_i' satisfies MMP$\langle m_i, n, k_i, s\rangle$, where $i \in [1, x]$. This implies that, every column in M_i' has at least $s + 1$ cells

Table II. The Output Matrix of Algorithm $Mark(M, k, s)$, When $m = 4$, $n = 6$, $k = 3$ and $s = 1$

	C_1	C_2	C_3	C_4	C_5	C_6
R_1	1	1	1	0	0	0
R_2	1	0	0	1	1	0
R_3	0	1	0	1	0	1
R_4	0	0	1	0	1	1

marked with 1, and no fewer than k_i rows together have a cell marked with 1 in every column.

We integrate $M'_1, \cdots, M'_x$ into an $m \times n$ matrix M' to form a marked version of M. Initially, every cell in M' is marked with 0. Since $\Sigma_{j=1}^{x} m_j = m$ and $\Sigma_{j=1}^{x} n_j = n$, we can place $M'_1, \cdots, M'_x$ into M' in such a way that no pair of these x matrices share a row or a column in M'. According to the properties of $M'_1, \cdots, M'_x$, at least $\Sigma_{i=1}^{x} k_i = k$ rows are required to have a cell marked with 1 in every column of M'. Also, since $M'_1, \cdots, M'_x$ together spans all columns of M' (because $\Sigma_{j=1}^{x} n_j = n$) and there are at least $s+1$ cells marked with 1 in every column of M'_i ($i \in [1, x]$), every column of M' has at least $s+1$ cells marked with 1. In general, M' satisfies MMP$\langle m, n, k, s+1\rangle$. □

Note that when $k \leq n$, we can always find a set of x integers $\{k_1, \cdots, k_x\}$ such that $k = \Sigma_{i=0}^{x} k_i$ and $n \geq \Sigma_{i=1}^{x} C_{s+1}^{k_i+s}$. In the extreme case, let $x = k$ and we have $k_1 = \cdots = k_x = 1$. In this case, $\Sigma_{i=1}^{x} C_{s+1}^{k_i+s} = \Sigma_{i=1}^{k} C_{s+1}^{s+1} = k \leq n$. That is to say, Lemma 22 applies regardless of the value of n and k.

The following theorem improves the upper-bound (k+xs) given by Lemma 22.

THEOREM 23. *Given k and s, let x and y be two integers such that $x \in [1, k]$ and $y \in [1, (s+1)]$. Let $d_x = \lfloor k/x \rfloor$, $r_x = k$ mod x, $d_y = \lfloor (s+1)/y \rfloor$ and $r_y = (s+1)$ mod y.* MMP$\langle m, n, k, s+1\rangle$ *is satisfiable if the following conditions hold.*

$$\bullet\ n \geq (x-1)C_{d_y+r_y}^{d_x+d_y+r_y-1} + C_{d_y+r_y}^{d_x+r_x+d_y+r_y-1} \quad (1)$$

$$\bullet\ m \geq yk + x(s+1) - xy \quad (2)$$

The proof to this theorem is in Appendix C. The intuition is that, in addition to "splitting" k into $k_1, \cdots, k_x$ as stated in Lemma 22, we can "split" $(s+1)$ into $s_1, \cdots, s_y$ as well to acquire a tighter upper-bound. Our proof makes use of the following facts that are derived from Definition 8.

Fact 1. If MMP$\langle m, n, k, x\rangle$ is satisfiable, then MMP$\langle m-x', n, k, x-x'\rangle$ is satisfiable, where $x' \in [0, x)$.

Fact 2. If both MMP$\langle m_1, n, k, x_1\rangle$ and MMP$\langle m_2, n, k, x_2\rangle$ are satisfiable, then MMP$\langle m_1+m_2, n, k, x_1+x_2\rangle$ is satisfiable.

Let m_o denote the smallest value of m such that MMP$\langle m, n, k, s+1\rangle$ is satisfiable. The upper-bound of m_o given by Theorem 23 is the minimum value of $yk + x(s+1) - xy$, provided that Condition (1) is satisfied. Since $x \in [1, k]$ and $y \in [1, s+1]$, we need to try at most $k(s+1)$ combinations of x and y before finding the minimum value. In Figure 5, we compare the upper-bound given by Theorem 23 with the exact value of m_o computed by a brute-force algorithm

(n,k,s)	$(3,2,2)$	$(4,3,2)$	$(4,3,3)$	$(5,3,3)$	$(6,3,3)$	$(8,3,3)$	$(12,3,3)$
Thm 23	5	8	10	10	8	8	8
Exact	5	8	10	9	8	7	7

Fig. 5. Comparing the upper-bound given in Theorem 23 with the exact value of m_o. All listed cases have $n < C_{s+1}^{s+k}$.

which tries all possible ways to mark matrix M[2]. We can see that the upper-bound given in Theorem 23 is close to the exact value in the cases we tested. Recall that it is unclear whether we can solve MMP (or ReSAT) in polynomial-time. Theorem 23 essentially presents an $O(k(s+1))$-time algorithm which gives an approximated answer to the problem.

7. RELATED WORK

To our knowledge, there is no prior work in resiliency policies in the context of access control. Prior analysis work in access control deals mostly with safety and security analysis, and separation of duty.

Simple safety analysis, that is, determining whether an access control system can reach a state in which an unsafe access is allowed, was first formalized by Harrison et al. [1976] in the context of the well-known access matrix model [Graham and Denning 1972; Lampson 1971], and was shown to be undecidable in the HRU model [Harrison et al. 1976]. Following that, there have been various efforts in designing access control systems in which simple safety analysis is decidable or efficiently decidable, for example, the take-grant model [Lipton and Snyder 1977], the schematic protection model [Sandhu 1988a], and the typed access matrix model [Sandhu 1992]. Koch et al. [2002a] considered safety in RBAC with the RBAC state and state-change rules posed as a graph formalism [Koch et al. 2002b]. Li et al. [2005] proposed the notion of security analysis which generalizes safety analysis; it was considered in the context of a trust management framework. Security analysis has since been considered also in the context of RBAC [Li and Tripunitara 2004].

Separation of duty (SoD) has long existed in the physical world, sometimes under the name "the two-man rule," for example, in the banking industry and the military. To our knowledge, in the information security literature the notion of SoD first appeared in Saltzer and Schroeder [1975] under the name "separation of privilege." Clark and Wilson's commercial security policy for integrity [1987] identified SoD along with well-formed transactions as two major mechanisms of fraud and error control. Separation of Duty policies were also studied in Ahn and Sandhu [2000], Crampton [2003], Gligor et al. [1998], Jaeger and Tidswell [2001], Li et al. [2004], Nash and Poland [1990], Sandhu [1990; 1988b], and Simon and Zurko [1997].

Another related concept is availability policies in Li et al. [2005] and Li and Tripunitara [2004], which asks whether a user always possesses certain permissions across state changes. In that work, checking whether an availability policy is satisfied in a state is straightforward; the challenges arises

[2]We do not have the results for larger parameters as the brute-force algorithm does not scale.

from the fact that the access control state may be changed by administrative operations, and the possible state space may be infinite. Unlike availability policies, resiliency policies such as the ones we consider in this article do not specify a permission requirement on any individual user; rather, they specify requirements about tolerating absent users and the overall ability of groups of users to perform critical tasks. Consequently, resiliency policies are more powerful and checking whether a state satisfies a resiliency policy is a challenging problem in itself.

Following the preliminary version of this paper, Wang and Li [2007] studied resiliency in workflow authorization systems. They proposed three levels of resiliency in workflow systems, namely, static resiliency, decremental resiliency, and dynamic resiliency.

8. DISCUSSIONS AND OPEN PROBLEMS

To our knowledge, this is the first work in access control research to clearly formulate properties on enabling access, rather than restricting access. Because this work opens up a new area, even though we have presented a number of results in this article, many more interesting problems remain open. One fruitful area of future research lies in the interaction between resiliency policies and other policies. In addition to resiliency and separation of duty policies, other kinds policies may exist. For example, an *assignment range policy* states that a set of permissions can be possessed only by a certain set of users. This may be motivated by the fact that not all users are qualified to receive these permissions. For example, the permission to install software on campus-wide network servers may be assigned only to qualified and authorized staff, and should not be given to others. The interaction among resiliency policies, SSoD policies, and assignment range policies is an interesting and challenging problem for future work.

Another open area lies in designing techniques to enforce resiliency policies. The resiliency checking problem studied in this paper essentially assumes that the permission authorization for any user does not change, even if some users become absent. With such a static approach, an access control state satisfies a resiliency policy only if the state contains enough redundancy on users and user-permission authorization. A dynamic approach may allow us to enforce resiliency policies with less resources. Delegation, which allows a user to act on another user's behalf, is such an approach. When a user is absent, some of her permissions can be temporarily delegated to one or more other users so that the remaining users still have enough permissions to complete the critical tasks. Naturally, we may require delegation activities satisfy separation of duty policies when applicable. The following example illustrates the ideal of using delegation to enforce resiliency policies.

Example 4. Task T requires two permissions p_1 and p_2. In order to prevent frauds, an SoD policy $\mathsf{sod}\langle\{p_1, p_2\}, 2\rangle$ has been specified, which indicates that a user can only have either p_1 or p_2. Thus, at least four users are needed for an access control state to satisfy the resiliency policy $\mathsf{rp}\langle\{p_1, p_2\}, 1, 1, \infty\rangle$, if user-permission authorization cannot be changed dynamically.

However, with delegation, only three users are required to enforce $\mathsf{rp}\langle\{p_1, p_2\}, 1, 1, \infty\rangle$ without violating the SoD policy. Let *Alice*, *Bob*, and *Carl* be three users in the access control state. *Alice* is given p_1, *Bob* is given p_2, while *Carl* has neither p_1 nor p_2. We have a delegation rule stating that when *Alice* is absent but *Bob* is available, p_1 is made available to *Carl* (until *Alice* comes back). Similarly, we have another delegation rule stating that when *Bob* is absent but *Alice* is available, p_2 is available to *Carl*. It is easy to see that the access control state with the two delegation rules can tolerate the absence of any one of the three users.

How to design delegation rules to effectively enforce resiliency policies with limited number of users is an open research problem that is of both theoretical and practical interests.

Finally, the computational complexity of m-ReSAT is unknown.

9. CONCLUSION AND FUTURE WORK

We have introduced the notion of resiliency policies in the context of access control systems. Unlike most existing work on policy analysis in access control, resiliency policies are about enabling access rather than restricting access. Resiliency policies are particularly useful when evaluating whether the access control configuration of a system is ready for emergency response. To the best of our knowledge, such resiliency policies have not been previously studied in access control.

We have shown that the problem of checking whether an access control state satisfies a resiliency policy in the general case is intractable (**NP**-hard), and is in the Polynomial Hierarchy (in $\mathbf{coNP^{NP}}$). We have shown also that several subcases of the problem remain intractable. Notwithstanding these intractability results, many instances that will be encountered in practice may be efficiently solvable. In an effort to seek an efficient solution for practical instances of the problem and to understand what the hard instances are, we have designed and implemented an algorithm for RCP. Our algorithm takes advantages of an effective static pruning approach and the existence of fast SAT solvers. Our experimental results have shown that the algorithm is capable to solve RCP instances of nontrivial sizes in a reasonable amount of time. We have also explored the co-existence of resiliency policies with static separation of duty (SSoD) policies. In particular, we have presented several computational complexity results on checking whether a set of resiliency policies and SSoD policies are consistent. Finally, we have combined the notion of resiliency and separation of duty to introduce the resilient separation of duty policy, which is useful in situations where both fault-tolerance and fraud-prevention are desired. We have studied the satisfaction problem as well as the optimization problem of resilient separation of duty policy.

APPENDIX A. BACKGROUND ON ORACLE TURING MACHINES AND POLYNOMIAL HIERARCHY

Oracle Turing Machines. An oracle Turing machine, with oracle L, is denoted as M^L. L is a language. M^L can use the oracle to determine whether a string

is in L or not in one step. More precisely, M^L is a two-tape deterministic Turing machine. The extra tape is called the oracle tape. M^L has three additional states: $q_?$ (the query state), and q_{yes} and q_{no} (the answer states). The computation of M^L proceeds like in any ordinary Turing machine, except for transitions from $q_?$. When M^L enters $q_?$, it checks whether the contents of the oracle tape are in L. If so, M^L moves to q_{yes}. Otherwise, M^L moves to q_{no}. In other words, M^L is given the ability to "instantaneously" determine whether a particular string is in L or not.

Polynomial Hierarchy. The polynomial hierarchy provides a more detailed way of classifying NP-hard decision problems. The complexity classes in this hierarchy are denoted by $\Sigma_k\mathbf{P}$, $\Pi_k\mathbf{P}$, $\Delta_k\mathbf{P}$, where k is a nonnegative integer. They are defined as follows:

$\Sigma_0\mathbf{P} = \Pi_0\mathbf{P} = \Delta_0\mathbf{P} = \mathbf{P}$,
and for all $k \geq 0$,
$\Delta_{k+1}\mathbf{P} = \mathbf{P}^{\Sigma_k\mathbf{P}}$,
$\Sigma_{k+1}\mathbf{P} = \mathbf{NP}^{\Sigma_k\mathbf{P}}$,
$\Pi_{k+1}\mathbf{P} = \mathbf{co}\text{-}\Sigma_{k+1}\mathbf{P} = \mathbf{coNP}^{\Sigma_k\mathbf{P}}$.

Some classes in the hierarchy are

$\Delta_1\mathbf{P} = \mathbf{P}$, $\Sigma_1\mathbf{P} = \mathbf{NP}$, $\Pi_1\mathbf{P} = \mathbf{coNP}$,
$\Delta_2\mathbf{P} = \mathbf{P}^{\mathbf{NP}}$, $\Sigma_2\mathbf{P} = \mathbf{NP}^{\mathbf{NP}}$,
$\Pi_2\mathbf{P} = \mathbf{coNP}^{\mathbf{NP}}$.

APPENDIX B. METHODOLOGY FOR GENERATING TESTING INSTANCES

Our goals of implementing the algorithm and performing experiments are to understand the effectiveness of the pruning techniques developed in Section 4 and to understand how well the algorithm scales with different parameters. To achieve such goals, we try to generate instances to approximate realistic instances. We generate instances for testing using combinations of the following approaches.

—*Purely Random*. For each permission p_i and user u_j, we assign p_i to u_j with a certain probability. The probability is an adjustable parameter which is called the *density* parameter.

—*With Constraints*. Often times, an access control system may include (explicit or implicit) constraints that restrict user-permission assignment. For example, there may be a requirement that no user is authorized for permissions p_i and p_j at the same time. To model this aspect, mutual exclusion constraints among permissions are randomly generated. Two permissions are mutually exclusive if no user can be authorized for both permissions. The total number of pairs of permissions is $p(p-1)/2$. The number of constraints to be generated is determined by an adjustable parameter that specifies the ratio of the constraints to $p(p-1)/2$. After the generation of constraints and user-permission assignment, if a user is assigned to two permissions

that are mutually exclusive, we randomly remove one permission from the assignment.

—*Density Variation*: In situations where resiliency is an issue, it is likely that some permissions are assigned only to a small number of people. To model these situations, we assign different permissions with different densities. We have two parameters that specify the lower bound and the upper bound for the permission assignment densities respectively. The sequence of all permissions $p_1, \cdots, p_m$ will be assigned with nondecreasing density, with p_1 being assigned with the lower bound density and p_m with the upper bound density.

Finally, if a user is not assigned any permission, we randomly assign one permission to the user, so that we do not have a useless user in the generated instance.

APPENDIX C. PROOFS IN SECTION 6

Proof of Lemma 20

We need to prove that the matrix generated by $Mark(M, k, s)$ (see Figure 4) satisfies MMP$\langle m, n, k, s+1\rangle$. From the description of the algorithm, each column contains exactly $s+1$ cells marked with 1's. What remains to show is that there does not exist a set of $k-1$ rows who together have a cell marked with 1 in every column. Without loss of generality, we consider the case where $n = C^{k+s}_{s+1}$.

Let M' be the marked version of M generated by $Mark(M, k, s)$. We prove by induction on s and k.

Based Case 1. When $s = 0$, we have $n = C^{k+s}_{s+1} = C^k_1 = k$ and $m = k + s = k$. $E^k_1 = \{(1), (2), \cdots, (k)\}$. According to the algorithm, each of the k rows contains exactly one cell marked with 1. The statement holds.

Based Case 2. When $k = 1$, the statement trivially holds.

Inductive Case. Assume that the statement is true for $Mark(M, k-1, s)$ and $Mark(M, k, s-1)$, where $k \geq 2$ and $s \geq 1$. We need to prove that it remains true for $Mark(M, k, s)$.

According to the algorithm, the number of cells that are marked with 1 in each row is C^{k+s-1}_s. In particular, in Row 1, cells $M(1, 1), \cdots, M(1, C^{k+s-1}_s)$ are marked with 1. Let $M\langle i_1, j_1, i_2, j_2\rangle$ denote the submatrix of M whose top-left cell is $M(i_1, j_1)$ and bottom-right cell is $M(i_2, j_2)$.

First of all, we show that Row 1 plus any other $k-2$ rows together do not have cells marked with 1 in every column. According to the algorithm, the submatrix $M\langle 2, C^{k+s-1}_s + 1, k+s, C^{k+s}_{s+1}\rangle$ is identical to M_b, where M_b is a $(k+s-1) \times C^{k+s-1}_{s+1}$ matrix marked by $Mark(M_b, k-1, s)$ (See Table III, cells marked with '1_b'). By induction hypothesis, no $k-2$ rows in M_b together have a cell marked with 1 in every column of M_b. Therefore, Row 1 plus any other $k-2$ rows in M together do not have a cell marked with 1 in every column of M.

Table III. The Output of $Mark(M, 3, 2)$.

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
R_1	1	1	1	1	1	1				
R_2	1_a	1_a	1_a				1_b	1_b	1_b	
R_3	1_a			1_a	1_a		1_b	1_b		1_b
R_4		1_a		1_a		1_a	1_b		1_b	1_b
R_5			1_a		1_a	1_a		1_b	1_b	1_b

Note: Cells marked with "1_a" and "1_b" are the outputs of $Mark(M_a, 3, 1)$ and $Mark(M_b, 2, 2)$, respectively. Empty cells are marked with 0.

Second, we show that no $k-1$ rows other than Row 1 together have cells marked with 1 in every column. According to the algorithm, the submatrix $M\langle 2, 1, k+s, C_s^{k+s-1}\rangle$ is identical to M_a, where M_a is a $(k+s-1)\times C_s^{k+s-1}$ matrix marked by $Mark(M_a, k, s-1)$ (See Table III, cells marked with '1_a'). By induction hypothesis, no $k-1$ rows in M_a together have cells marked with 1 in every column.

In general, the statement holds.

We conclude that the algorithm $Mark(M, k, s)$ marks M in a way that satisfies MMP. Therefore, Lemma 20 holds.

Proof of Theorem 23

We prove, without loss of generality, that when $yk + x(s+1) - xy = m$ and $(x-1)C_{d_y+r_y}^{d_x+d_y+r_y-1} + C_{d_y+r_y}^{d_x+r_x+d_y+r_y-1} = n$, both MMP$\langle y(d_x-1)+s+1, C_{d_y+r_y}^{d_x+d_y+r_y-1}, d_x, s+1\rangle$ and MMP$\langle y(d_x+r_x-1)+s+1, C_{d_y+r_y}^{d_x+r_x+d_y+r_y-1}, d_x+r_x, s+1\rangle$ are satisfiable, where $d_x = \lfloor k/x \rfloor$, $r_x = k \bmod x$, $d_y = \lfloor (s+1)/y \rfloor$ and $r_y = (s+1) \bmod y$.

We start with MMP$\langle y(d_x-1)+s+1, C_{d_y+r_y}^{d_x+d_y+r_y-1}, d_x, s+1\rangle$. First of all, according to Lemma 20, MMP$\langle d_x+d_y+r_y-1, C_{d_y+r_y}^{d_x+d_y+r_y-1}, d_x, d_y+r_y\rangle$ is satisfiable. From Fact 1, MMP$\langle d_x+d_y-1, C_{d_y+r_y}^{d_x+d_y+r_y-1}, d_x, d_y\rangle$ is satisfiable as well. Note that $d_y \times (y-1) + d_y + r_y = s+1$. Also, $(d_x+d_y-1)\times(y-1) + d_x + d_y + r_y - 1 = y(d_x-1)+s+1$. Hence, from Fact 2, MMP$\langle y(d_x-1)+s+1, C_{d_y+r_y}^{d_x+d_y+r_y-1}, d_x, s+1\rangle$ is satisfiable.

Similarly, we can prove that MMP$\langle y(d_x+r_x-1)+s+1, C_{d_y+r_y}^{d_x+r_x+d_y+r_y-1}, d_x+r_x, s+1\rangle$ is satisfiable.

Let M_1 be the marked version of a $(y(d_x-1)+s+1)\times C_{d_y+r_y}^{d_x+d_y+r_y-1}$ matrix that satisfies MMP$\langle y(d_x-1)+s+1, C_{d_y+r_y}^{d_x+d_y+r_y-1}, d_x, s+1\rangle$, and M_2 be the marked version of a $(y(d_x+r_x-1)+s+1)\times C_{d_y+r_y}^{d_x+r_x+d_y+r_y-1}$ matrix that satisfies MMP$\langle y(d_x+r_x-1)+s+1, C_{d_y+r_y}^{d_x+r_x+d_y+r_y-1}, d_x+r_x, s+1\rangle$. Note that $(x-1)(y(d_x-1)+s+1) + y(d_x+r_x-1)+s+1 = yk + x(s+1) - xy = m$ and $(x-1)C_{d_y+r_y}^{d_x+d_y+r_y-1} + C_{d_y+r_y}^{d_x+r_x+d_y+r_y-1} = n$. Thus, we can place $x-1$ copies of M_1 and one copy of M_2 into an $m \times n$ matrix M, such that no pair of these x copies share a row or a column in M. Also, $d_x \times (x-1) + d_x + r_x = k$. According to the properties of M_1 and M_2, no less than k rows in M together

have a cell marked with 1 in every column of M. Furthermore, every column in M_1 and M_2 contains $s+1$ cells marked with 1. In general, M satisfies MMP$\langle m, n, k, s+1 \rangle$. The theorem holds.

REFERENCES

AHN, G.-J. AND SANDHU, R. S. 2000. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Sec. 3*, 4, 207–226.

CLARK, D. D. AND WILSON, D. R. 1987. A comparision of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'87)*. IEEE Computer Society Press, 184–194.

CRAMPTON, J. 2003. Specifying and enforcing constraints in role-based access control. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT'03)*. 43–50.

DU, D., GU, J., AND PARDALOS, P. M., Eds. 1997. Satisfiability problem: Theory and applications. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 35*. AMS Press.

GAREY, M. R. AND JOHNSON, D. J. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

GLIGOR, V. D., GAVRILA, S. I., AND FERRAIOLO, D. F. 1998. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of IEEE Symposium on Research in Security and Privacy (SP'98)*. 172–183.

GRAHAM, G. S. AND DENNING, P. J. 1972. Protection—Principles and practice. In *Proceedings of the American Federation of Information Processing Societies National Semiannual Computer Conference Spring Joint Computer Conference (AFIPS'72)*. 40, 417–429.

HARRISON, M. A., RUZZO, W. L., AND ULLMAN, J. D. 1976. Protection in operating systems. *Comm. ACM 19*, 8, 461–471.

JAEGER, T. AND TIDSWELL, J. E. 2001. Practical safety in flexible access control models. *ACM Trans. Inf. Syst. Sec. 4*, 2, 158–190.

KOCH, M., MANCINI, L. V., AND PARISI-PRESICCE, F. 2002a. Decidability of safety in graph-based models for access control. In *Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS'02)*. Springer, 229–243.

KOCH, M., MANCINI, L. V., AND PARISI-PRESICCE, F. 2002b. A graph-based formalism for RBAC. *ACM Trans. Inf. Syst. Sec. 5*, 3 (Aug.), 332–365.

LAMPSON, B. W. 1971. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems (CISS'71)*. (Reprinted in *ACM Operat. Syst. Rev. 8*, 1, 18–24).

LE BERRE, D. 2006. SAT4J: A satisfiability library for Java. Retrieved from http://www.sat4j.org/.

LI, N., BIZRI, Z., AND TRIPUNITARA, M. V. 2004. On mutually-exclusive roles and separation of duty. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'04)*. ACM Press, 42–51.

LI, N., MITCHELL, J. C., AND WINSBOROUGH, W. H. 2005. Beyond proof-of-compliance: Security analysis in trust management. *J. ACM 52*, 3, 474–514.

LI, N. AND TRIPUNITARA, M. V. 2004. Security analysis in role-based access control. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies (SACMAT'04)*. 126–135.

LIPTON, R. J. AND SNYDER, L. 1977. A linear time algorithm for deciding subject security. *J. ACM 24*, 3, 455–464.

NASH, M. J. AND POLAND, K. R. 1990. Some conundrums concerning separation of duty. In *Proceedings of IEEE Symposium on Research in Security and Privacy (SP'90)*. 201–209.

PAPADIMITRIOU, C. H. 1994. *Computational Complexity*. Addison Wesley Longman.

SALTZER, J. H. AND SCHROEDER, M. D. 1975. The protection of information in computer systems. *Proc. IEEE 63*, 9, 1278–1308.

SANDHU, R. 1990. Separation of duties in computerized information systems. In *Proceedings of the International Federation Information Processing WG11.3 Workshop on Database Security (IFIP'90)*.

SANDHU, R. S. 1988a. The schematic protection model: Its definition and analysis for acyclic attenuating systems. *J. ACM 35*, 2, 404–432.

SANDHU, R. S. 1988b. Transaction control expressions for separation of duties. In *Proceedings of the 4th Annual Computer Security Applications Conference (ACSAC'88)*.

SANDHU, R. S. 1992. The typed access matrix model. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'92)*. IEEE Computer Society Press, 122–136.

SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. 1996. Role-based access control models. *IEEE Comput. 29*, 2, 38–47.

SIMON, T. T. AND ZURKO, M. E. 1997. Separation of duty in role-based environments. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW'97)*. IEEE Computer Society Press, 183–194.

WANG, Q. AND LI, N. 2007. Satisfiability and resiliency in workflow systems. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'07)*.

Received May 2007; revised June 2008; accepted July 2008