Resiliency Policies in Access Control

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

Mahesh V. Tripunitara
Motorola Labs
tripunit@motorola.com

Qihua Wang
Dept. of Computer Science
Purdue University
wangq@cs.purdue.edu

ABSTRACT

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. Finally, we study the consistency problem between resiliency policies and static separation of duty policies.

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

Keywords

Access Control, Fault-tolerant, Policy Design

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 [9, 17, 22] studies whether an access right can be leaked to unauthorized users. Separation of duty (SoD) policies [2, 20] 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 principal, the senior treasurer has all permissions except the one for logging, while each of the junior treasurers has only one of the three permissions. Since 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. 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.

Our contributions in this paper 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 $\mathbf{coNP^{NP}}$, 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 non-trivial 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 [14]. We study the policy consistency problem between resiliency policies and SSoD policies. 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, but is in $\mathbf{NP^{NP}}$.

The remainder of this paper 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. We discuss related work in Section 6. Finally, we conclude and present open problems related to the concept of resiliency in Section 7.

2. RESILIENCY POLICIES AND THE RESILIENCY CHECKING PROBLEM

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

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

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.

Example 1. Consider the access control state 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 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.

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 ∞.

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 2 users, there exist 3 mutually disjoint sets of users such that each set covers all permissions in P, then after removing any 3 users, there are at least 2 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 3 users *Alice*, *Bob* and *Carl* that possess *Endorse* to belong to distinct sets; this still leaves one permission that needs to 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 [13, 8] and Role Based Access Control systems [25]. 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 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
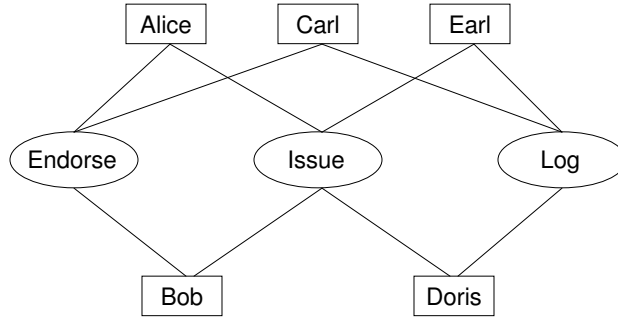
Figure 1: An example of an access control state with 5 users, *Alice, Bob, Carl, Doris* **and** *Earl*, **and 3 permissions,** *Endorse, Issue* **and** *Log*. **A line segment connects a user (e.g.,** *Alice*) **to a permission (e.g.,** *Endorse*) **to indicate that the user has the permission. This 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.**

cases where some of the three parameters are degenerated. For example, a resiliency policy in the subcase $\mathsf{RCP}\langle s = 0, d = 1\rangle$ has the form $\mathsf{rp}(P, 0, 1, t)$, which asks whether there exists a set of users of size at most t that together have all permissions in P; while the subcase $\mathsf{RCP}\langle t = \infty\rangle$ asks whether there exist 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, $\mathsf{RCP}\langle\ \rangle$ 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 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 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 ver-

ifiable 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 A for a brief introduction to the Polynomial Hierarchy.) Because the Polynomial Hierarchy collapses when $\mathbf{P = NP}$, 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 $\mathbf{P = NP}$.

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 [19] (also referred to as MINIMUM COVERING problem in [6]) 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
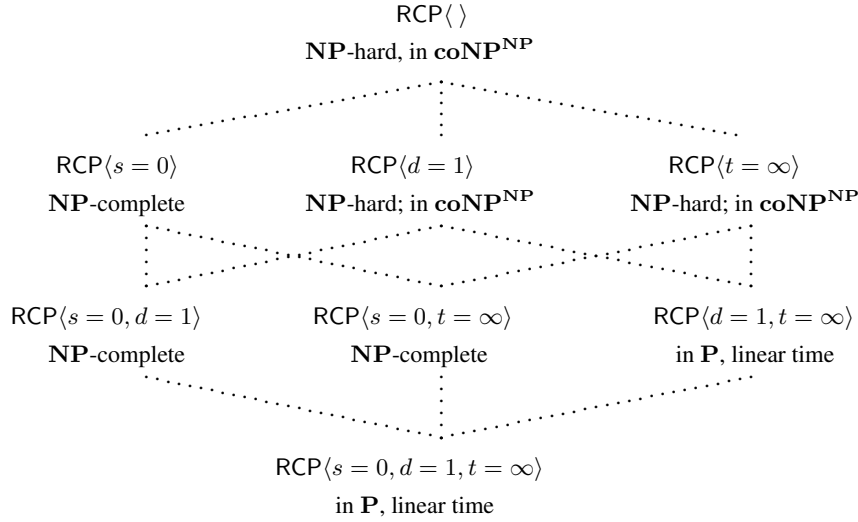
$$\mathsf{RCP}\langle\,\rangle$$
NP-hard, in **coNP$^{\mathbf{NP}}$**

| $\mathsf{RCP}\langle s=0\rangle$ | $\mathsf{RCP}\langle d=1\rangle$ | $\mathsf{RCP}\langle t=\infty\rangle$ |
| **NP**-complete | **NP**-hard; in **coNP$^{\mathbf{NP}}$** | **NP**-hard; in **coNP$^{\mathbf{NP}}$** |

| $\mathsf{RCP}\langle s=0, d=1\rangle$ | $\mathsf{RCP}\langle s=0, t=\infty\rangle$ | $\mathsf{RCP}\langle d=1, t=\infty\rangle$ |
| **NP**-complete | **NP**-complete | in **P**, linear time |

$$\mathsf{RCP}\langle s=0, d=1, t=\infty\rangle$$
in **P**, linear time

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

that K subsets cover S. Then, the K users to which the subsets are mapped together have all 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 [6] 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 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* **coNP$^{\mathbf{NP}}$**.

PROOF. We show that the complement of $\mathsf{RCP}\langle\,\rangle$ is in **NP$^{\mathbf{NP}}$**. 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 **NP$^{\mathbf{NP}}$**. Therefore, $\mathsf{RCP}\langle\,\rangle$ is in **coNP$^{\mathbf{NP}}$**.

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

An instance in $\mathsf{RCP}\langle d=1, t=\infty\rangle$ asks whether an access control state satisfies a policy $\mathsf{rp}\langle P, s, 1, \infty\rangle$. We observe that the answer is "no" if and only if some permission in P is assigned to no more than s users. In this case, removing the s users who have that permission would result in no user having that permission. On the other hand, if each permission is assigned to at least $s+1$ users,

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 permissions in P.

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 at least $s+1$. More generally, 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". 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.

We now give a linear-time algorithm for calculating the tolerance bound. This, together with the above observations, suffices to prove Lemma 6. The algorithm maintains a counter for each permission. It first goes through all pairs in UP to count how many users each permission is assigned to. It then returns the minimal value among the counters.

4. AN ALGORITHM FOR RCP

The fact that RCP is intractable (**NP**-hard) 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. We now describe an algorithm for RCP. We first describe an algorithm for the subcase $\mathsf{RCP}\langle t=\infty\rangle$, i.e., 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 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 rp $\langle P, s, d, \infty \rangle$. Our algorithm adds the following improvements, which greatly reduces the running time.

Preprocessing Given the state UP and the policy 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".

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, [5]), 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 ensure 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.

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 dominates 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 of only users not in A_2, (2) they are mutually disjoint, and (3) users in each set together have all 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.

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. Secondly, we use an index e that initially has 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.

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.

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 [4], 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 [4], 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 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 around 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.

The algorithm scales reasonably well with n when d is small; however when d is over about 8, the algorithm stops scaling. The running time of the algorithm depends on the total number of absent sets that need 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 running time of the algorithm for cases in which the instance is true, for increasing n (number of users) and d. 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 running time of our algorithm. This threshold of around 8 seems to hold for many other experiments we have performed.

Static pruning is very effective Table 1 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 of static pruning is increasingly pronounced for larger values of n
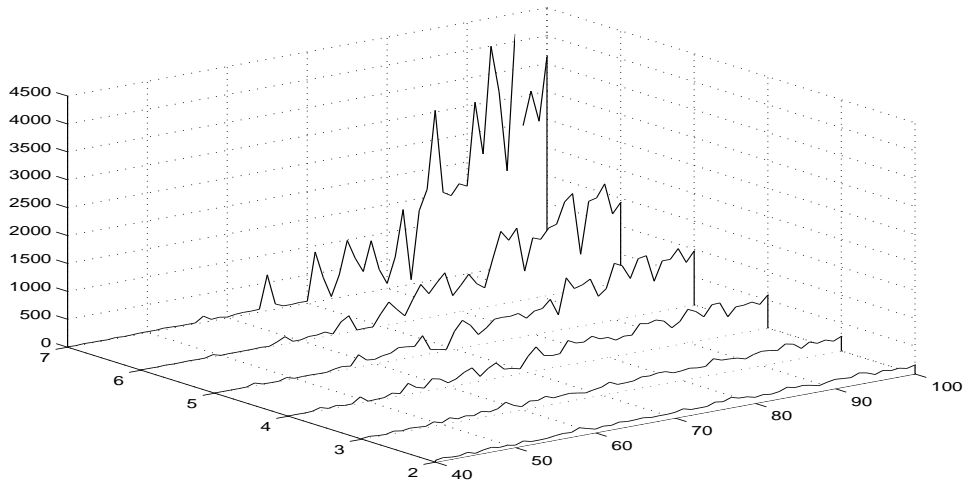
Figure 3: This graph shows the effect on running time (in milliseconds) as the number of users n and the number of disjoint sets d increase. The size of absent sets is 3 and there are 10 permissions. The value of n increases from 40 to 100 and the value of d increases from 2 to 7. For smaller values of n (say, $n = 40$), increasing d has almost no effect on the running time 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 running time.

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 $s = 4, n = 40$ (1042 absent sets).

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, i.e., 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 co-existence of resiliency policies with 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. [14]. 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 can 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 can be performed. It is clear that neither kind of policies 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).

119

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

Table 1: A table that shows that static pruning is effective. 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., $\binom{n}{s}$). We observe that the effect of static pruning is pronounced, especially for large values of n and s. There is always an improvement of at least 1 order of magnitude, and when $n = 100$ and $s = 8$, there is an improvement of 7 orders of magnitude.

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 $\mathsf{PCCP}\langle F \rangle$. 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 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 $\mathbf{NP^{NP}}$.

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 $\langle m, n \rangle$, *i.e., the most general case of* PCCP, *is in* $\mathbf{NP^{NP}}$.

The proof for Theorem 11 is in Appendix C. It is of course possible that there are special cases of PCCP that are efficiently solvable. Detailed analysis of the tractable subcases of PCCP is beyond the scope of this paper.

6. 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, i.e., determining whether an access control system can reach a state in which an unsafe access is allowed, was first formalized by Harrison et al. [9] in the context of the well-known access matrix model [8, 13], and was shown to be undecidable in the HRU model [9]. Following that, there have been various efforts in designing access control systems in which simple safety analysis is decidable or efficiently decidable, e.g., the take-grant model [17], the schematic protection model [22], and the typed access matrix model [24]. Koch et al. [11] considered safety in RBAC with the RBAC state and state-change rules posed as a graph formalism [12]. Li et al. [15] 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 [16].

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 [20] under the name "separation of privilege." Clark and Wilson's commercial security policy for integrity [2] identified SoD along with well-formed transactions as two major mechanisms of fraud and error control. Separation of Duty policies were also studied in [1, 3, 7, 10, 14, 18, 21, 23, 26].

Another related concept is availability policies in [15, 16], 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 from the fact that the access control state may be changed by administra-

tive operations, and the possible state space may be infinite. Unlike availability policies, resiliency policies such as the ones we consider in this paper 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.

7.   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.

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 paper, many more interesting problems remain open. One fruitful area of future research lies in the interaction between resiliency policies and other policies. In the study of the consistency problem with SSoD policies and resiliency policies, we do not consider the total number of available users as a limiting factor. In practice, the number of users in any organization is bounded. This makes it harder to satisfy both resiliency policies (which require users to possess more permissions) and SSoD policies (which require users to possess fewer permissions). Hence, it would be interesting to consider the consistency problem with an upper bound on the number of users in the access control state.

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 for enforcing resiliency policies: if a state does not satisfy an existing set of policies, how do we alter the state to make it satisfy these policies? This problem seems to be particularly interesting in the context of Role-Based Access Control systems, where one changes the role assignments of users to satisfy existing policies. Another approach for achieving resiliency is to use delegation; that is, when a user is absent, some of his permissions can be automatically and temporarily assigned to one or more other users. However, we may require such delegation to satisfy other coexisting policies such as separation of duty.

Acknowledgement

8.   REFERENCES

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

[2] D. D. Clark and D. R. Wilson. A comparision of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, May 1987.

[3] J. Crampton. Specifying and enforcing constraints in role-based access control. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, pages 43–50, Como, Italy, June 2003.

[4] Daniel Le Berre (project leader). SAT4J: A satisfiability library for Java. URL http://www.sat4j.org/, Jan. 2006.

[5] D. Du, J. Gu, and P. M. Pardalos, editors. *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS Press, 1997.

[6] M. R. Garey and D. J. Johnson. *Computers And Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[7] V. D. Gligor, S. I. Gavrila, and D. F. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 172–183, May 1998.

[8] G. S. Graham and P. J. Denning. Protection — principles and practice. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 40, pages 417–429. AFIPS Press, May 16–18 1972.

[9] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, Aug. 1976.

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

[11] M. Koch, L. V. Mancini, and F. Parisi-Presicce. Decidability of safety in graph-based models for access control. In *Proceedings of the Seventh European Symposium on Research in Computer Security (ESORICS 2002)*, pages 229–243. Springer, Oct. 2002.

[12] M. Koch, L. V. Mancini, and F. Parisi-Presicce. A graph-based formalism for RBAC. *ACM Transactions on Information and System Security*, 5(3):332–365, Aug. 2002.

[13] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, 1971. Reprinted in ACM Operating Systems Review, 8(1):18-24, Jan 1974.

[14] N. Li, Z. Bizri, and M. V. Tripunitara. On mutually-exclusive roles and separation of duty. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS-11)*, pages 42–51. ACM Press, Oct. 2004.

[15] N. Li, J. C. Mitchell, and W. H. Winsborough. Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM*, 52(3):474–514, May 2005. Preliminary version appeared in *Proceedings of 2003 IEEE Symposium on Security and Privacy*.

[16] N. Li and M. V. Tripunitara. Security analysis in role-based access control. In *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies (SACMAT 2004)*, pages 126–135, June 2004.

[17] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM*, 24(3):455–464, 1977.

[18] M. J. Nash and K. R. Poland. Some conundrums concerning separation of duty. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 201–209, May 1990.

[19] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley Longman, 1994.

[20] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[21] R. Sandhu. Separation of duties in computerized information systems. In *Proceedings of the IFIP WG11.3 Workshop on Database Security*, Sept. 1990.

[22] R. S. Sandhu. The schematic protection model: Its definition and analysis for acyclic attenuating systems. *Journal of the ACM*, 35(2):404–432, 1988.

[23] R. S. Sandhu. Transaction control expressions for separation of duties. In *Proceedings of the Fourth Annual Computer Security Applications Conference (ACSAC'88)*, Dec. 1988.

[24] R. S. Sandhu. The typed access matrix model. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, pages 122–136. IEEE Computer Society Press, May 1992.

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

[26] T. T. Simon and M. E. Zurko. Separation of duty in role-based environments. In *Proceedings of The 10th Computer Security Foundations Workshop*, pages 183–194. IEEE Computer Society Press, June 1997.

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

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

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 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 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.

C. PROOFS FOR THEOREM 11

Without loss of generality, we assume that for any static separation of duty policy $\mathsf{ssod}\langle P, k\rangle$, 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 [19] (also referred to as MINIMUM COVERING problem in [6]) 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 $S = \mathsf{ssod}\langle P, b+1\rangle$ 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 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 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 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$\langle F\rangle$. 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 $S_i = \mathsf{ssod}\langle P_i, 2\rangle$ $(1 \leq i \leq m)$, 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 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 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* $S_i = \mathsf{ssod}\langle P_i, k_i\rangle$ $(1 \leq i \leq m)$ *and* $R_j = \mathsf{rp}\langle Q_j, s_j, d_j, t_j\rangle$ $(1 \leq j \leq n)$. *PCCP*$\langle F\rangle$ *is in* $\mathbf{NP^{NP}}$.

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 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 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 to check whether UP violates any SSoD policy. (In order to prove that a state violates a static separation of duty policy $\mathsf{ssod}\langle P, k\rangle$, we just need to present a set of no more than k users in the state who together have all 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 $\mathbf{NP^{NP}}$ in general.