Policy Analysis for Administrative Role Based Access Control without Separate Administration

Ping Yang
Department of Computer Science, State University of New York at Binghamton, USA

Mikhail I. Gofman
Department of Computer Science, California State University at Fullerton, USA

Scott D. Stoller
Department of Computer Science, Stony Brook University, USA

Zijiang Yang
Department of Computer Science, Western Michigan University, USA

Abstract

Role based access control (RBAC) is a widely used approach to access control with well-known advantages in managing authorization policies. This paper considers user-role reachability analysis of administrative role based access control (ARBAC), which defines administrative roles and specifies how members of each administrative role can change the RBAC policy. Most existing works on user-role reachability analysis assume the separate administration restriction in ARBAC policies. While this restriction greatly simplifies the user-role reachability analysis, it also limits the expressiveness and applicability of ARBAC. In this paper, we consider analysis of ARBAC without the separate administration restriction and present new techniques to reduce the number of ARBAC rules and users considered during analysis. We also present parallel algorithms that speed up the analysis on multi-core systems. The experimental results show that our techniques significantly reduce the analysis time, making it practical to analyze ARBAC without separate administration.

1 Introduction

Role based access control (RBAC) [2] is a widely used approach to access control with well-known advantages in performing authorization management. An RBAC policy is a tuple $\langle U, R, P, UA, PA \rangle$ where U, R and P are finite sets of users, roles, and permissions, respectively, $UA \subseteq U \times R$ is the user-role assignment relation, and $PA \subseteq P \times R$ is the permission-role assignment relation. RBAC also supports role hierarchy: $r_1 \succeq r_2$ specifies that r_1 is senior to r_2 (or r_2 is junior to r_1), which implies that every member of r_1 is also a member of r_2, and every permission assigned to r_2 is also available to members of r_1.

Administrative role-based access control'97 (ARBAC'97) [17] defines administrative roles and specifies how members of each administrative role can change the RBAC policy. One part of ARBAC specifies *user-role administration* which controls changes to user-role assignments. The policy is expressed by two types of rules: (1) $can\_assign(r_a, c, r_t)$ grants an administrative role r_a permission to assign a target role r_t to any user who satisfies the precondition c, and (2) $can\_revoke(r_a, r_t)$ grants an administrative role r_a permission to revoke a target role r_t from a user. The precondition c is a conjunction of literals, where each literal is either r (positive precondition) or $\neg r$ (negative precondition) for some role r. A user satisfies a positive (negative) precondition if the user is (is not) a member of the role. A role is called an *administrative role* if it has an administrative permission, i.e., if it appears in the first component of a can_assign or a can_revoke rule. ARBAC'97 requires *separate administration* [22], *i.e.*, administrative roles cannot be target roles in *can_assign* and *can_revoke* rules or appear in preconditions. In the rest of this paper, we represent the precondition c as $P \wedge \neg N$, where P contains all positive preconditions in c and N contains all negative preconditions in c.

The correctness of ARBAC policies is critical to system security, because any errors in ARBAC may result in violations of confidentiality or integrity. In large organizations with complex ARBAC policies, manual inspection of ARBAC policies for correctness can be impractical, because actions performed by different administrators may interfere with each other in subtle ways. Thus, automated analysis algorithms are essential to ensure that an ARBAC policy conforms to desired correctness properties.

This paper considers user-role reachability analysis of ARBAC [22], which asks queries of the form "given an RBAC policy ϕ, an ARBAC policy ψ, a set of users U, a target user u_t, and a set of roles (called the "goal"), is it possible for users in $U \cup \{u_t\}$ to assign u_t to all roles in the goal?". Several other security analysis problems, such as user-role availability [16], role containment [16], and weakest precondition [22], can be reduced to this problem.

User-role reachability analysis is intractable even under various restrictions on the ARBAC policy [16, 18]. Most existing research on user-role reachability analysis [9, 8, 14] follows the definition of ARBAC'97, which assumes separate administration. This prohibits an administrative role to serve as the target role in ARBAC rules, so it is sufficient to consider the user-role assignments of only the target user. However, in practice, the separate administration restriction does not always hold. For example, a university ARBAC policy may specify that the role $DeptChair$ can assign a member of role $Faculty$ to role $AdmissionsComittee$, which can in turn assign any user to role $Student$. Formally, this policy is expressed by rules $can\_assign(DeptChair,$ $\{Faculty\} \wedge \neg\emptyset$, $AdmissionsComittee)$ and $can\_assign(AdmissionsCommittee,$ $\emptyset \wedge \neg\emptyset$, $Student)$, which do not satisfy the separate administration restriction.

Analysis of ARBAC without separate administration is significantly more challenging, because it must consider administrative actions that change the role memberships of all users, not only the target user. For example, a non-target user u may assign another non-target user u_1 to an administrative role, which can in turn change the role assignments of the target user. Stoller et al. [22] tackled this problem by developing an algorithm that is fixed parameter tractable with respect to the numbers of users and mixed roles (i.e. roles that appear both negatively and positively in the policy). This means that the algorithm is exponential in the numbers of users and mixed roles, but is polynomial in the size of the policy when the numbers of users and mixed roles are fixed. Since the number of users is large in large organizations, and the algorithm is exponential in the number of users, the algorithm does not scale well. For example, we used the implementation in [22] to analyze a university ARBAC policy containing 150 users, and the program failed to terminate within 6 hours for 4 out of the 8 randomly generated queries.

Contributions This paper presents a number of reduction techniques that improve the scalability of the algorithm in [22]. Our main contributions are summarized below.

- We propose two static reduction techniques – enhanced slicing (Section 3.1) and hierarchical rule reduction (Section 3.6) – to reduce the number of ARBAC rules considered during analysis.
- We develop a user equivalent set reduction technique (Sections 3.3) and several involved-user reduction techniques (Section 3.4) to reduce the number of users considered during the analysis.
- We propose two lazy reduction techniques – the subsumed-user reduction and the delayed revocation reduction – to delay performing unnecessary transitions (Section 3.5).
- We present several parallel algorithms, which speed up the analysis on multi-core or multi-processor platforms (Section 4).
- We evaluated the effectiveness of our reduction techniques and our parallel algorithms on an ARBAC policy for university administration. The experimental results show that our techniques significantly reduce the analysis time.

Organization The rest of the paper is organized as follows. Section 2 describes the user-role reachability analysis algorithm for ARBAC without separate administration developed in [22]. Sections 3 and 4 present our reduction techniques and parallel algorithms, respectively. The experimental results are given in Section 5, followed by a discussion of related research in Section 6. Section 7 concludes the paper.

2 Preliminaries: User-Role Reachability Analysis of ARBAC

User-role reachability queries are introduced in Section 1. Let UA_0 be a set of all user-role assignments in the RBAC policy ϕ. The user-role reachability query is represented as a tuple $Q = \langle UA_0, U, u_t, \psi, goal \rangle$.

Stoller et al. [22] presented an algorithm for analyzing ARBAC without separate administration, which is formalized in Algorithm 1. A role is *negative* if it appears negatively in some precondition in the policy; otherwise, it is *non-negative*. A role is *positive* if it appears in the goal, appears positively in some precondition in the policy, or is an administrative role; otherwise, it is *non-positive*. A role that is both negative and positive is a *mixed* role. Note that their algorithm applies to ARBAC without role hierarchy; role hierarchy can be eliminated using the algorithm in [18]. The algorithm works as follows.

The algorithm first performs a slicing transformation (function $slicing$ in Line 3), which back-chains along ARBAC rules to identify roles that are possibly useful for assigning to users. Such roles are preconditions and administrative roles, directly or indirectly (i.e., transitively), for assigning the target user to any goal roles. Lines 27–30 identify can_assign rules that are possibly useful for assigning those roles to users, add their positive preconditions to Rel_+, and add their negative preconditions to Rel_-. Lines 31–34 identify possibly useful can_revoke rules, i.e. can_revoke rules that revoke roles in Rel_-, and can_assign rules for assigning users to administrative roles in these can_revoke rules. The algorithm then computes a set $RelRule$ of all rules that are possibly useful for assigning the target user to any role in the goal; such rules are called "relevant rules". $RelRule$ includes all can_assign rules whose targets are in Rel_+ and all can_revoke rules whose targets are in Rel_-.

Next, the algorithm constructs a *reduced state graph* G using rules in $RelRule$. Each state in G is a set of user-role assignments, and each transition describes an allowed change to the state defined by the ARBAC policy ψ. A transition is either $ua(r_a, u, r)$ which specifies that an administrative role r_a adds user u to role r, or $ur(r_a, u, r)$ which specifies that an administrative role r_a revokes user u from role r. The following reductions are applied: (1) Transitions that revoke non-negative roles (i.e., roles in $Rel_+ \setminus Rel_-$) or add non-positive roles (i.e., $Rel_- \setminus Rel+$) are prohibited because they do not enable any other transitions; (2) Transitions that add non-negative roles or revoke non-positive roles are *invisible*; such transitions will not disable any other transitions. Transitions that add or revoke mixed roles are *visible*. Invisible transitions together with a visible transition form a single composite transition.

The graph G is constructed as follows. First, the algorithm computes $\mathrm{closure}(UA_0)$, which is the largest state that is reachable from UA_0 by performing all invisible transitions enabled from UA_0 (function $closure$ in Line 3). The algorithm then computes a set of all states reachable from $\mathrm{closure}(UA_0)$ (Lines 5–21), and returns true iff there exists a state s in G such that $goal \subseteq \{r \mid (u_t, r) \in s\}$ (Lines 4, 11, and 18). The worst-case time complexity for constructing the graph is $O(2^{(|MR||U|)}|I|^c)$ for some constant c, where $|MR|$ is the number of mixed roles, $|U|$ is the number of users, and $|I|$ is the size of the policy [22].

A condition called the *hierarchical role assignment (HRA)* is defined in [22], under which analysis of ARBAC without separate administration can be reduced to analysis of ARBAC with separate administration. An ARBAC policy satisfies HRA if, for all $can\_assign(r_a, P \wedge \neg N, r)$ where r is an administrative role, $r_a \succeq r$.

Example 1 *Consider the following ARBAC policy ψ, and the reachability query for this policy with the initial RBAC policy $UA_0 = \{(u_1, r_1), (u_1, r_3), (u_2, r_2),$*

Algorithm 1 The User-Role Reachability Analysis Algorithm in [22].

```
1:  $Processed = Rel_+ = Rel_- = \emptyset$; $RelRule = \emptyset$;
2:  **procedure** $analysis(UA_0, U, u_t, \psi, goal)$
3:  $(Rel_+, Rel_-, RelRule) = slicing(UA_0, \psi, goal)$;
4:  $W = Reached = \{closure(U, u_t, UA_0)\}$;
5:  **if** $goal \subseteq \{r \mid (u_t, r) \in closure(U, u_t, UA_0)\}$ **then** return true; **end if**
6:  **while** $W \neq \emptyset$ **do**
7:      remove a state $s$ from $W$;
8:      **for all** (user $u \in (U \cup \{u_t\})$) **do**
9:        **for all** $can\_assign(r_a, P \wedge \neg N, r) \in RelRule$ **do**
10:         **if** ($r \in (Rel_+ \cap Rel_-)$, $(u, r) \notin s$, $P \subseteq \{r \mid (u, r) \in s\}$, $N \cap \{r \mid (u, r) \in s\} = \emptyset$,
            and $(u', r_a) \in s$ for some user $u'$)
11:         **then** $s' =$ closure$(U, u_t, s \cup \{(u, r)\})$; add transition $s \overset{ua(r_a,u,r)}{\rightarrow} s'$ to $G$;
12:             **if** $goal \subseteq \{r|(u_t, r) \in s'\}$ **then** return true; **end if**
13:             **if** $s' \notin Reached$ **then** $W = W \cup \{s'\}$; $Reached = Reached \cup \{s'\}$; **end if**
14:         **end if**    **end for**   **end for**
15:     **for all** ($can\_revoke(r_a, r) \in RelRule$)
16:       **for all** (user $u \in (U \cup \{u_t\})$)
17:         **if** ($(u, r) \in s$ and $(u', r_a) \in s$ for some user $u'$)
18:         **then** $s' =$ closure$(U, u_t, s \setminus \{(u, r)\})$; add transition $s \overset{ur(r_a,u,r)}{\rightarrow} s'$ to $G$;
19:             **if** $goal \subseteq \{r \mid (u_t, r) \in s'\}$ **then** return true; **end if**
20:             **if** $s' \notin Reached$ **then** $W = W \cup \{s'\}$; $Reached = Reached \cup \{s'\}$; **end if**
21:         **end if**    **end for**   **end for**
22: **end while**
23: return false;
24: **procedure** $slicing(UA_0, \psi, goal)$
25: **if** $goal = \emptyset$ **then** return $(\emptyset, \emptyset, \emptyset)$ **end if**
26: $Processed = Processed \cup goal$;   $R_+ = goal$;   $R_- = \emptyset$;   $Rule = \emptyset$;
27: **for all** $can\_assign(r_a, P \wedge \neg N, r) \in \psi$ where $r \in goal$ **do**
28:     $(R_1, R_2, R_3) = slicing(UA_0, \psi, (\{r_a\} \cup P) \setminus Processed)$;   $R_+ = R_+ \cup R_1$;
29:     $R_- = R_- \cup N \cup R_2$;   $Rule = Rule \cup \{can\_assign(r_a, P \wedge \neg N, r)\} \cup R_3$;
30: **end for**
31: $RelRev = \{can\_revoke(r_a, r) \in \psi \mid r \in R_-\}$;   $Rule = Rule \cup RelRev$;
32: **for all** $can\_revoke(r_a, r) \in RelRev$ where $r_a \notin Processed$
33: $(R_4, R_5, R_6) = slicing(UA_0, \psi, \{r_a\})$;   $R_+ = R_+ \cup R_4$;
34: $R_- = R_- \cup R_5$;   $Rule = Rule \cup R_6$; **end for**
35: return $(R_+, R_-, Rule)$;
36: **procedure** $closure(U, u_t, s)$
37: $s_1 = s$;
38: **for all** $can\_assign(r_a, P \wedge \neg N, r) \in RelRule$ **do**
39:   **for all** user $u \in (U \cup \{u_t\})$ **do**
40:     **if** ($r \in (Rel_+ \setminus Rel_-)$, $(u, r) \notin s$, $P \subseteq \{r \mid (u, r) \in s\}$, $N \cap \{r \mid (u, r) \in s\} = \emptyset$, and
        $(u', r_a) \in s$ for some user $u'$)
41:     **then** $s_1 = s_1 \cup (u, r)$;    **end if**   **end for**   **end for**
42: **if** $s == s_1$ **then** return $s_1$; **else** return $closure(U, u_t, s_1)$;
```

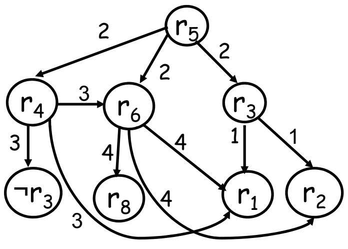


Figure 1: The graph illustrating the slicing process in Example 1.

(u_2, r_8), (u_3, r_2), (u_3, r_8), $(u_t, r_6)\}$, the target user u_t, the set of non-target users $U = \{u_1, u_2, u_3\}$, and the goal $\{r_5\}$.

1. can_assign$(r_1, \{r_2\} \wedge \neg\emptyset, r_3)$	*2. can_assign*$(r_6, \{r_4, r_3\} \wedge \neg\emptyset, r_5)$
3. can_assign$(r_1, \{r_6\} \wedge \neg\{r_3\}, r_4)$	*4. can_assign*$(r_2, \{r_8, r_1\} \wedge \neg\emptyset, r_6)$
5. can_assign$(r_2, \{r_6\} \wedge \neg\emptyset, r_7)$	*6. can_revoke*(r_1, r_2)
7. can_revoke(r_1, r_3)	*8. can_revoke*(r_1, r_4)

This policy does not satisfy the separate administration restriction, because role r_6 is an administrative role in rule 2 and a target role in rule 4.

First, the algorithm performs slicing to compute Rel_+ and Rel_-. Initially, Rel_+ contains all roles in the goal, i.e. r_5, and Rel_- is empty. Figure 1 illustrates the slicing process. Each edge in the figure is labeled with the number of the can_assign rule applied during slicing. Roles in nodes without $\neg$ are positive preconditions or administrative roles of the can_assign rule applied; such roles are added to Rel_+. Roles in nodes containing $\neg$ are negative preconditions of the can_assign rule applied; such roles are added to Rel_-. The set of mixed roles is computed as $Rel_+ \cap Rel_- = \{r_3\}$. The algorithm then computes the set of rules $RelRule$ that are useful for assigning the target user to any role in the goal. $RelRule$ contains all can_assign rules whose target roles are in Rel_+ and can_revoke rules whose target roles are in Rel_-, i.e., rules 1, 2, 3, 4, and 7.

Next, the algorithm computes the initial state $closure(UA_0)$ and all states reachable from $closure(UA_0)$ using rules in $RelRule$. Because $(u_t, r_6) \in UA_0$ and $(u_t, r_3) \notin UA_0$, rule 3 is applied to assign r_4 to u_t. Since r_4 is non-negative, (u_t, r_4) is added to $closure(UA_0)$ through an invisible transition. Because $(u_2, r_2) \in UA_0$, $(u_3, r_2) \in UA_0$, and r_3 is a mixed role, rule 1 is applied to assign r_3 to u_2 and u_3 through visible transitions $ua(r_1, u_2, r_3)$ and $ua(r_1, u_3, r_3)$. Similarly, because $(u_1, r_3) \in UA_0$ and r_3 is a mixed role, rule 7 is applied to revoke r_3 from u_1 through a visible transition $ur(r_1, u_1, r_3)$. The algorithm stops when no more transitions can be computed. The resulting graph appears in Figure 2. Because the graph does not contain (u_t, r_5), the goal is not reachable. $\square$

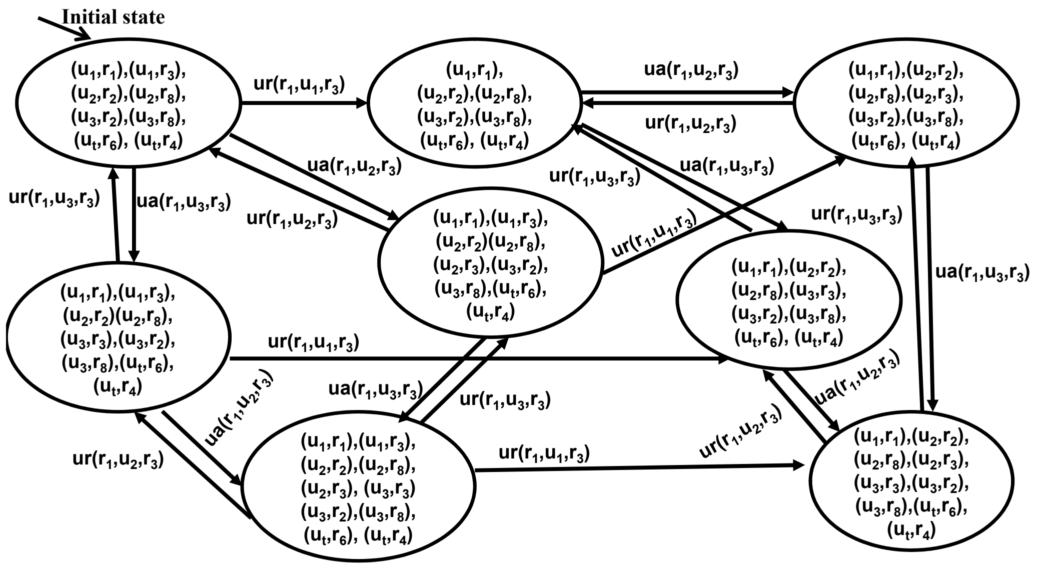

Initial state
(u1,r1),(u1,r3), (u2,r2),(u2,r8), (u3,r2),(u3,r8), (ut,r6), (ut,r4)
ur(r1,u1,r3)
(u1,r1), (u2,r2),(u2,r8), (u3,r2),(u3,r8), (ut,r6), (ut,r4)
ua(r1,u2,r3)
ur(r1,u2,r3)
(u1,r1),(u2,r2), (u2,r8),(u2,r3), (u3,r2),(u3,r8), (ut,r6), (ut,r4)
ua(r1,u2,r3)
ur(r1,u2,r3)
ur(r1,u3,r3)
ua(r1,u3,r3)
ur(r1,u3,r3)
ua(r1,u3,r3)
(u1,r1),(u1,r3), (u2,r2)(u2,r8), (u2,r3),(u3,r2), (u3,r8),(ut,r6), (ut,r4)
ur(r1,u1,r3)
ur(r1,u3,r3)
ua(r1,u3,r3)
(u1,r1),(u2,r2), (u2,r8),(u3,r3), (u3,r2),(u3,r8), (ut,r6), (ut,r4)
(u1,r1),(u1,r3), (u2,r2)(u2,r8), (u3,r3),(u3,r2), (u3,r8),(ut,r6), (ut,r4)
ur(r1,u1,r3)
ua(r1,u3,r3)
ur(r1,u3,r3)
ua(r1,u2,r3)
ur(r1,u2,r3)
(u1,r1),(u1,r3), (u2,r2),(u2,r8), (u2,r3), (u3,r3) (u3,r2),(u3,r8), (ut,r6), (ut,r4)
ur(r1,u1,r3)
ur(r1,u2,r3)
ua(r1,u2,r3)
(u1,r1),(u2,r2), (u2,r8),(u2,r3), (u3,r3),(u3,r2), (u3,r8),(ut,r6), (ut,r4)

Figure 2: The state graph constructed in Example 1 using the algorithm in [22].

3 Reduction Techniques

The analysis algorithm described in Section 2 does not scale well for policies containing a large number of users. Let $Q = \langle UA_0, U, u_t, \psi, goal \rangle$ be a user-role reachability query. This section presents several techniques for reducing the numbers of users and ARBAC rules considered during analysis.

3.1 Enhanced Slicing

This section presents an enhanced slicing algorithm to further reduce the number of roles and rules processed during slicing. Our enhanced slicing has the following features. First, because the target user needs to be assigned to all roles in the goal, while non-target users do not need to be assigned to any role in the goal, the enhanced slicing treats these two types of users differently. As a result, the size of the sliced policy for non-target users may be smaller than that for the target user. Second, our enhanced slicing for the target user eliminates can_assign rules whose negative preconditions contain an irrevocable role assigned to the target user in the initial policy, because such rules can never be enabled. Third, our enhanced slicing does not process non-negative and irrevocable administrative roles assigned to some user in the initial RBAC policy, because such roles can never be revoked and hence do not need to be reassigned to any user.

Enhanced slicing for the target user We say that a role r is *irrevocable* in an ARBAC policy if the policy does not contain a can_revoke rule with target r. Algorithm 2 gives our enhanced slicing algorithm for the target user u_t.

The algorithm first identifies roles that are possibly useful for assigning to u_t. Such

Algorithm 2 An Enhanced Slicing Algorithm for the Target User.

```
1:  procedure $targetslicing(UA_0, u_t, \psi, goal)$
2:  if $goal = \emptyset$ then return $(\emptyset, \emptyset, \emptyset)$; end if
3:  $Processed = Processed \cup goal$;   $R_+ = goal$;   $R_- = \emptyset$;   $Rule = \emptyset$;
4:  for all $can\_assign(r_a, P \wedge \neg N, r) \in \psi$ where $r \in goal$ do
5:      if $N$ does not contain an irrevocable role $r'$ such that $(u_t, r') \in UA_0\}$ then
6:          $S = \{r \mid r \in (P \cup \{r_a\}) \wedge (r$ is nonnegative or irrevocable$) \wedge (u_t, r) \in UA_0\}$;
7:          $(R_1, R_2, R_3) = targetslicing(UA_0, \psi, ((\{r_a\} \cup P) \setminus S) \setminus Processed)$;
8:          $R_+ = R_+ \cup R_1$;   $R_- = R_- \cup N \cup R_2$;
9:          $Rule = Rule \cup \{can\_assign(r_a, P \wedge \neg N, r)\} \cup R_3$;
10:     end if
11: end for
12: $RelRev = \{can\_revoke(r_a, r) \in \psi \mid r \in R_-\}$;   $Rule = Rule \cup RelRev$;
13: for all $can\_revoke(r_a, r) \in RelRev$ where $r_a \notin Processed$ do
14:     $S = \{r_a \mid (r_a$ is nonnegative or irrevocable$) \wedge (u_t, r_a) \in UA_0\}$;
15:     $(R_4, R_5, R_6) = slicing(UA_0, \psi, \{r_a\} \setminus S)$;   $R_+ = R_+ \cup R_4$;
16:     $R_- = R_- \cup R_5$;   $Rule = Rule \cup R_6$;
17: end for
18: return $(R_+, R_-, Rule)$;
```

roles are preconditions and administrative roles, directly or indirectly, for assigning u_t to any role in the goal. Lines 4–11 identify can_assign rules that are possibly useful for assigning those roles to u_t, add their positive preconditions and administrative roles to Rel_+, and add their negative preconditions to Rel_-. The differences between our enhanced slicing algorithm and the slicing algorithm in [22] are: (1) if administrative roles or positive preconditions of the aforementioned can_assign rules are non-negative or irrevocable and have already been assigned to u_t in the initial RBAC policy UA_0, then we do not add them to Rel_+ and do not process them during slicing. This is safe because such roles will not be revoked during analysis and hence there is no need to reassign them to u_t (Lines 6–7); (2) for every aforementioned can_assign rule, we check if the negative precondition of the rule contains an irrevocable role assigned to u_t in UA_0. If so, such a rule can never be enabled and hence will not be applied during slicing (Line 5).

Lines 12–17 identify can_revoke rules that are possibly useful for assigning the target user to any role in the goal and can_assign rules that enable administrative roles of those can_revoke rules. Such rules include can_revoke rules that revoke roles in Rel_- and can_assign rules that are possibly useful for assigning the target user to administrative roles of can_revoke rules identified. Similar to can_assign rules, if administrative roles of can_revoke rules identified above are non-negative or irrevocable and are already assigned to the target user, then we do not process such roles during slicing (Lines 14–15). In addition, since a negative role may become non-negative after slicing, to further reduce the number of relevant rules computed, we perform slicing multiple times until the set of all negative roles remains unchanged.

Algorithm 3 An Enhanced Slicing Algorithm for Non-target Users.

```
1: $Processed = \emptyset$;
2: **procedure** $nontarslicing(UA_0, u_t, \psi, goal)$
3: **if** ($goal == \emptyset$) **then** return $(\emptyset, \emptyset, \emptyset)$; **end if**
4: $Processed = Processed \cup goal$;   $R_+ = R_- = \emptyset$;   $Rule = \emptyset$;
5: **for all** $can\_assign(r_a, P \wedge \neg N, r)$ where $(u_t, r) \in goal$ **do**
6:     **if** $N$ does not contain an irrevocable role $r'$ such that $(u_t, r') \in UA_0$ **then**
7:         **if** ($(u, r_a) \in UA_0$ for some user $u$ and ($r_a$ is non-negative or irrevocable)) **then**
8:             $R_1 = R_2 = R_3 = \emptyset$;
9:         **else** $(R_1, R_2, R_3) = slicing(UA_0, \psi, \{r_a\})$; **end if**
10:        $S = \{r \mid r \in P \wedge$ ($r$ is non-negative or irrevocable) $\wedge (u_t, r) \in UA_0\}$;
11:        $(R'_1, R'_2, R'_3) = nontarslicing(UA_0, \psi, (P \setminus S) \setminus Processed)$;
12:        $R_+ = R_+ \cup S \cup R_1 \cup R'_1$; $R_- = R_- \cup R_2 \cup R'_2$; $Rule = Rule \cup R_3 \cup R'_3$;
13:    **end if**
14: **end for**
15: $RelRev = \{can\_revoke(r_a, r) \mid r \in R_-\}$;   $Rule = Rule \cup RelRev$;
16: **for all** $can\_revoke(r_a, r) \in RelRev$ **do**
17:     **if** $r_a \notin Processed \wedge$ ($r_a$ is negative $\vee r_a$ is a non-negative role not assigned to any user
        in $UA_0$) **then**
18:         $(R_4, R_5, R_6) = slicing(UA_0, \psi, \{r_a\} \setminus Processed)$;
19:         $R_+ = R_+ \cup R_4$;   $R_- = R_- \cup R_5$;   $Rule = Rule \cup R_6$;
20:     **end if**
21: **end for**
22: return $(R_+, R_-, Rule)$;
```

Enhanced slicing for non-target users The algorithm tries to assign non-target users to administrative roles with permission to assign the target user u_t to the goal, and to roles useful for that purpose. For example, consider the following ARBAC policy:

1.$can\_assign(r_1, \{r_3, r_4\} \wedge \neg\emptyset, r_5)$
2.$can\_assign(r_2, \{r_1, r_6\} \wedge \neg\emptyset, r_3)$
3.$can\_assign(r_2, \{r_7, r_8\} \wedge \neg\emptyset, r_4)$

When analyzing whether the target user u_t can be assigned to all roles in the goal $\{r_5\}$, it is not useful to assign non-target users to r_5. As a result, it is not useful to assign non-target users to roles in the positive precondition of rule 1, i.e., r_3 and r_4. Instead, it is sufficient to assign non-target users to the administrative role r_1 which has permission to assign u_t to r_5, and administrative role r_2 which has permission to assign u_t to r_3 and r_4 (i.e., roles in the positive precondition of rule 1).

Algorithm 3 gives our enhanced slicing algorithm for non-target users. Lines 5–21 identify can_assign and can_revoke rules that are possibly useful for assigning the target user to any role in the goal, add administrative roles of such rules to Rel_+, and apply the slicing algorithm in [22] to process administrative roles. The differences between our enhanced slicing algorithm for non-target users and the slicing algorithm in [22] are: (1) our enhanced slicing algorithm does not add roles in the precondition

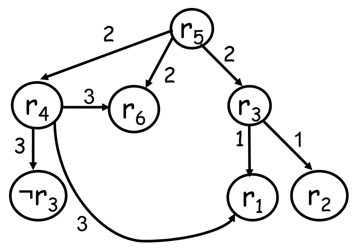


Figure 3: The graph illustrating the enhanced slicing for the target user in Example 2.

of the aforementioned can_assign rules to Rel_+ since it is not necessary to assign such roles to non-target users; (2) if administrative roles of the aforementioned can_assign rules identified above are non-negative or irrevocable and have already been assigned to some user in the initial RBAC policy UA_0, then we do not add them to Rel_+ and do not process them during slicing (Lines 7–8); (3) for each of the aforementioned can_assign rules, if the negative precondition of the rule contains an irrevocable role assigned to the target user in UA_0, then such a rule can never be enabled and hence will not be applied during slicing (Line 6).

Two-stage slicing While computing the initial state using the sliced policy, non-negative administrative roles that are not in the initial RBAC policy UA_0 may be added to the initial state $closure(UA_0)$ through invisible transitions. It is unnecessary to assign such roles to other non-target users during analysis if they do not appear in the positive precondition of the sliced policy. To further reduce the number of rules applied during the analysis, we apply the enhanced slicing algorithm twice: before and after computing the initial state. Our experiments on a university policy show that the two-stage slicing sometimes significantly reduces the number of relevant rules computed.

Example 2 *Consider Example 1. Figure 3 illustrates the enhanced slicing for the target user. The enhanced slicing is performed from role r_5 (i.e., the role in the goal). Since r_5 is the target role of rule 2, roles in the positive precondition of rule 2, i.e., r_3 and r_4, are added to Rel_+. Since the administrative role r_6 of rule 2 is a non-negative role and $(u_t, r_6) \in UA_0$, r_6 is not processed during slicing. The algorithm then performs enhanced slicing for the target user u_t from r_3 and r_4, adds r_1 and r_2 to Rel_+, and adds r_3 to Rel_-. Therefore, for the target user, $Rel_+ = \{r_1, r_2, r_3, r_4, r_5\}$, $Rel_- = \{r_3\}$, and $RelRule = \{1, 2, 3, 7\}$. Next, the algorithm performs enhanced slicing for non-target users. It is sufficient to assign non-target users to administrative roles that have permissions to assign the target user to the goal, i.e., r_1 and r_6. Since r_1 and r_6 are non-negative, $(u_1, r_1) \in UA_0$, and $(u_t, r_6) \in UA_0$, these roles are not processed during slicing. As a result, for non-target users, $Rel_+ = Rel_- = \emptyset$ and $RelRule = \emptyset$. This means that there is no need to assign roles to non-target users. The reduced state graph constructed with the enhanced slicing algorithm contains only one state $\{(u_1, r_1)$, (u_1, r_3), (u_2, r_2), (u_2, r_8), (u_3, r_2), (u_3, r_8), (u_t, r_6), $(u_t, r_4)\}$.*

3.2 Dead-role Reduction

We say that a role r is *dead* if r is not assigned to any user in the initial RBAC policy UA_0 and cannot be assigned to any user during the analysis. Our dead-role reduction aims to eliminate rules that are not useful for reaching the goal because they involve dead roles. The reduction first computes a set dr of roles that are not target roles of any can_assign rule and are not assigned to any user in UA_0; such roles are dead. Next, the reduction eliminates can_assign and can_revoke rules whose administrative roles are in dr, and can_assign rules whose positive preconditions are in dr, because such rules are never enabled. This process is repeated until no rules can be eliminated. The dead-role reduction may turn negative roles to non-negative, and hence may reduce the size of the state graph.

3.3 User Equivalent Set Reduction

In this section, we show that from each state it is sufficient to perform visible transitions for the target user and for non-target users assigned distinct sets of roles. Our technique is based on a notion of user equivalent set. The user equivalent set representation of a state s is basically an alternative representation of s, in which all users assigned the same set of roles are grouped together.

Definition 1 *The user equivalent set representation of a state s is defined as $ue(s)$ = $\{(Uset_1, Rset_1), \ldots, (Uset_n, Rset_n)\}$ where $Rset_1 \neq \ldots \neq Rset_n$, $Uset_1 \cup \ldots \cup Uset_n = \{u|(u,r) \in s\}$, and for every $u \in Uset_i$, $Rset_i = \{r|(u,r) \in s\}$.*

Let G_{ue} be the transition graph constructed using the user equivalent set representation. There is a transition $ue(s) \overset{A}{\rightarrow} ue(s')$ in G_{ue} if and only if there is a transition $s \overset{A}{\rightarrow} s'$ in G. The goal is reachable in G_{ue} if and only if there exists a state $s_g \in G_{ue}$ and $(Uset, Rset) \in s_g$ such that $u_t \in Uset$ and $goal \subseteq Rset$.

Our *user equivalent set reduction* works as follows. For every state s and every $(Uset, Rset) \in s$, we compute only transitions for the target user and transitions for **one** randomly selected non-target user in $Uset$, if $Uset$ contains such users. This is different from Algorithm 1, which computes transitions for **all** users in $Uset$. Intuitively, the user equivalent set reduction is correct because transitions performed on all users in $Uset$ are the same, and transitions performed on one user in $Uset$ do not disable transitions performed on other users in $Uset$. We use G_{redue} to denote the transition graph constructed with the user equivalent set reduction.

Given two states s_1 and s_2, we say that $s_1 \equiv s_2$ if there exists a substitution $\delta = \{u_1/u'_1, \ldots, u_n/u'_n\}$, where $u_1 \neq \ldots \neq u_n \neq u_t$ and $u'_1 \neq \ldots \neq u'_n \neq u_t$, such that $s_1\delta = s_2$. Theorem 2 formalizes the correctness of the reduction. We prove a lemma and then Theorem 2.

Lemma 1 *Let $Q = \langle UA_0, U, u_t, \psi, goal\rangle$ be a user-role reachability query, $s_0 = ue(closure(UA_0))$, and G_{redue} and G_{ue} be reduced state graphs constructed for Q with and without the user equivalent set reduction, respectively. (a) For every sequence* *of transitions* $s_0 \overset{A_0}{\to} s_1 \overset{A_1}{\to} \dots \overset{A_{n-2}}{\to} s_{n-1} \overset{A_{n-1}}{\to} s_n$ *in* G_{ue}*, there exists a sequence of transitions* $s_0 \overset{A'_0}{\to} s'_1 \overset{A'_1}{\to} \dots \overset{A'_{n-2}}{\to} s'_{n-1} \overset{A'_{n-1}}{\to} s'_n$ *in* G_{redue}*, where* $s_i \equiv s'_i$*,* $A_i = \alpha(r_a, u_i, r)$*,* $A'_i = \alpha(r_a, u'_i, r)$*,* $\alpha = ua/ur$*, and there exist* $(Uset, Rset) \in s_i$ *and* $(Uset', Rset') \in s'_i$ *such that* $Rset = Rset'$*,* $u_i \in Uset$*, and* $u'_i \in Uset'$*. (b) For every sequence of transitions* $s_0 \overset{A_1}{\to} s_1 \overset{A_1}{\to} \dots \overset{A_n}{\to} s_n$ *in* G_{redue}*, there exists a sequence of transitions* $s_0 \overset{A_1}{\to} s_1 \dots \overset{A_{n-1}}{\to} s_n$ *in* G_{ue}*.*

PROOF: We prove part (a) by induction on the length n.

Base Case: When $n = 0$, both G_{ue} and G_{redue} contain only one state $ue(closure(UA_0))$. The lemma holds.

Induction: Assume that the lemma holds when $n = k$. We prove the lemma for $n = k + 1$.

Let $s_0 \overset{A_1}{\to} s_1 \overset{A_2}{\to} \dots s_{k-1} \overset{A_k}{\to} s_k \overset{A_{k+1}}{\to} s_{k+1}$ be a sequence of transitions in G_{ue}. By the induction hypothesis, G_{redue} contains a sequence of transitions $s_0 \overset{A'_1}{\to} s'_1 \overset{A'_2}{\to} \dots s'_{k-1} \overset{A'_k}{\to} s'_k$, where $s_i \equiv s'_i$, $A_i = \alpha(r_a, u_i, r)$, $A'_i = \alpha(r_a, u'_i, r)$, $\alpha = ua/ur$, and there exist $(Uset, Rset) \in s_i$ and $(Uset', Rset') \in s'_i$ such that $Rset = Rset'$, $u_i \in Uset$, and $u'_i \in Uset'$.

If $A_{k+1} = ua(ra_{k+1}, u_t, r_{k+1})$, then there exists $(Uset, Rset) \in s_k$ such that $u_t \in Uset$. Because $s_k \equiv s'_k$, there exists $(Uset', Rset') \in s'_k$ such that $u_t \in Uset'$ and $Rset' = Rset$. Therefore, G_{redue} contains transition $s'_k \overset{A_{k+1}}{\to} s'_{k+1}$ and $s_{k+1} \equiv s'_{k+1}$. If $A_{k+1} = ua(ra_{k+1}, u_{k+1}, r_{k+1})$ where $u_{k+1} \neq u_t$, then there exists $(Uset, Rset) \in s_k$ such that $u_{k+1} \in Uset$. Because $s_k \equiv s'_k$, there exists $(Uset', Rset') \in s'_k$ such that $Rset' = Rset$. Let u'_{k+1} be a user in $Uset'$. Because u'_{k+1} and u_{k+1} are assigned the same set of roles, we can perform transition $s'_k \overset{ua(ra_{k+1}, u'_{k+1}, r_{k+1})}{\to} s'_{k+1}$ and $s_{k+1} \equiv s'_{k+1}$. The case where $A_{k+1} = ur(ra_{k+1}, u_{k+1}, r_{k+1})$ can be similarly proved. Thus, part(a) holds.

Part (b) holds because G_{redue} is a subgraph of G_{ue}. □

Theorem 2 *Let* $Q = \langle UA_0, U, u_t, \psi, goal \rangle$ *be a user-role reachability query, and* G_{redue} *and* G_{ue} *be reduced state graphs constructed for* Q *with and without the user equivalent set reduction, respectively. The goal is reachable in* G_{ue} *iff the goal is reachable in* G_{redue}*.*

PROOF: Suppose the goal is reachable in G_{ue}. Then there exists a sequence of transitions $s_0 \overset{A_0}{\to} s_1 \overset{A_1}{\to} \dots \overset{A_{n-1}}{\to} s_g$ in G_{ue} such that $(Uset, Rset) \in s_g$, $u_t \in Uset$, and $goal \subseteq Rset$. From Lemma 1(a), there exists a sequence of transitions $s_0 \overset{A'_0}{\to} s'_1 \overset{A'_1}{\to} \dots \overset{A'_{n-1}}{\to} s'_g$ in G_{redue} where $s_i \equiv s'_i$, $A_i = \alpha(r_a, u_i, r)$, $A'_i = \alpha(r_a, u'_i, r)$, $\alpha = ua/ur$, and there exist $(Uset, Rset) \in s_i$ and $(Uset', Rset') \in s'_i$ such that $Rset = Rset'$, $u_i \in Uset$, and $u'_i \in Uset'$. Because $s_g \equiv s'_g$, there exists $(Uset', Rset') \in s'_g$ such that $u_t \in Uset'$ and $Rset = Rset'$. As a result, $goal \subseteq Rset'$ and hence the goal is reachable in G_{redue}.

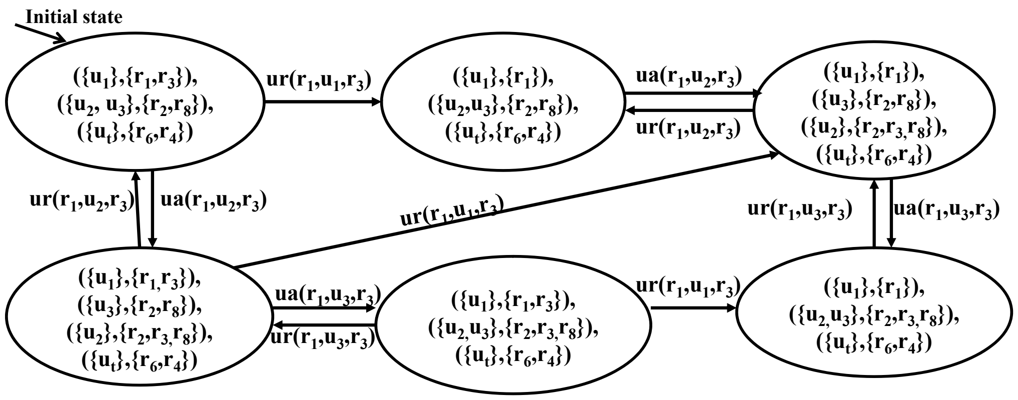

Initial state
({u1},{r1,r3}), ({u2, u3},{r2,r8}), ({ut},{r6,r4})
ur(r1,u1,r3)
({u1},{r1}), ({u2,u3},{r2,r8}), ({ut},{r6,r4})
ua(r1,u2,r3)
ur(r1,u2,r3)
({u1},{r1}), ({u3},{r2,r8}), ({u2},{r2,r3,r8}), ({ut},{r6,r4})
ur(r1,u2,r3)
ua(r1,u2,r3)
ur(r1,u1,r3)
ur(r1,u3,r3)
ua(r1,u3,r3)
({u1},{r1,r3}), ({u3},{r2,r8}), ({u2},{r2,r3,r8}), ({ut},{r6,r4})
ua(r1,u3,r3)
ur(r1,u3,r3)
({u1},{r1,r3}), ({u2,u3},{r2,r3,r8}), ({ut},{r6,r4})
ur(r1,u1,r3)
({u1},{r1}), ({u2,u3},{r2,r3,r8}), ({ut},{r6,r4})

Figure 4: The state graph constructed in Example 3 with the user equivalent set reduction.

Suppose G_{redue} contains a goal state s_g. Then from Lemma 1(b), s_g is also a state in G_{ue} and hence the goal is reachable. Therefore, the theorem holds. □

Example 3 *Consider the user-role reachability query in Example 1. Since non-target users u_2 and u_3 are assigned the same set of roles in the initial state, the algorithm with user equivalent set reduction performs only transitions for u_2 or u_3, but not both, from the initial state. In contrast, Algorithm 1 performs transitions for both u_2 and u_3 from the initial state. The graph constructed with the user equivalent set reduction is given in Figure 4.*

User-counter optimization In our implementation, we reduce the size of states by replacing $Uset$ in $(Uset, Rset)$ with a pair $(counter, target)$, where $counter$ records the number of non-target users in $Uset$, and $target$ is either 1 (indicating $u_t \in Uset$) or 0 (indicating $u_t \notin Uset$). This optimization may significantly reduce the size of states if many users are assigned the same set of roles. For example, in a university, thousands of users may be assigned the Student role.

Example 4 *Consider the user-role reachability query in Example 1 with an extended RBAC policy that assigns r_1 and r_3 to $u_1, u_4, \ldots, u_{1000}$, r_2 and r_8 to u_2 and u_3, and r_6 to u_t.*

Without the user-counter optimization, the initial state $init$ is $\{(\{u_1, u_4, \ldots, u_{1000}\}, \{r_1, r_3\}), (\{u_2, u_3\}, \{r_2, r_8\}), (\{u_t\}, \{r_6\})\}$, and the following two transitions are enabled from the initial state: $init \overset{ua(r_1,u_2,r_3)}{\rightarrow}$ $\{(\{u_1, u_4, \ldots, u_{1000}\}, \{r_1, r_3\}), (\{u_3\}, \{r_2, r_8\}), (\{u_2\}, \{r_2, r_3, r_8\}), (\{u_t\}, \{r_6\})\}$, and $init \overset{ur(r_1,u_1,r_3)}{\rightarrow}$ $\{(\{u_4, \ldots, u_{1000}\}, \{r_1, r_3\}), (\{u_1\}, \{r_1\}),$ $(\{u_2, u_3\}, \{r_2, r_8\}),$ $(\{u_t\}, \{r_6\})\}$.

With the user-counter optimization, the initial state $init$ is $\{((998, 0), \{r_1, r_3\}), ((2, 0), \{r_2, r_8\}), ((0, 1), \{r_6\})\}$, which specifies that 998 non-target users are assigned roles r_1 and r_3, 2 non-target users are assigned roles r_2 and r_8, and 1 target user is assigned role r_6. The fol-

lowing two transitions are enabled from the initial state: $init \overset{ua(r_1,0,r_3)}{\rightarrow}$ $\{((998,0),\{r_1,r_3\}),((1,0),\{r_2,r_8\}),(1,0),\{r_2,r_3,r_8\}),((0,1),\{r_6\})\}$ *and* $init \overset{ur(r_1,0,r_3)}{\rightarrow}$ $\{((997,0),\{r_1,r_3\}),((1,0),\{r_1\}),(2,0),\{r_2,r_8\}),((0,1),\{r_6\})\}$, *where* $0/1$ *in the transition specifies that the user is a non-target/target user.*

3.4 Involved-user Reduction

In Theorem 1 of [5], Ferrari et al. prove that, given a query, if the goal is reachable, then there exists a run (i.e. a finite sequence of transitions) in which the goal is reachable and at most $|AR| + 1$ users change their role-combinations, where AR is a set of administrative roles in the ARBAC policy and a role combination is the set of roles assigned to a user (we call it a role set). Based on this theorem, they propose to keep in the system at most $|AR| + 1$ users for each role set during analysis. In this section, we present three user-oriented reductions – static involved-user, spare-user, and dynamic involved-user reductions – that extend their work to further reduce the number of users considered during the analysis. The static involved-user reduction is performed prior to the construction of the state graph, the spare-user reduction is similar to the static involved-user reduction, except that it limits the number of non-target users in every state, not only the initial state, and the dynamic involved-user reduction is performed during the generation of the state graph.

Static involved-user reduction Our static involved-user reduction improves the reduction in [5] by computing a separate and sometimes smaller bound on the number of users needed for each role set (equivalently, for each set of roles in the user equivalent set representation) in the initial state, instead of a single bound that applies to all role sets in the initial state.

Let RS be a role set and $descendants(RS)$ be an upper-bound on the set of roles that can be assigned to users whose initial role set is RS. Formally, $descendants(RS)$ is defined as $LFP(\lambda S.RS \cup \{r \mid \text{there exists } can\_assign(r_a, P \wedge \neg N, r) \text{ in the ARBAC policy such that } P \subseteq S\})$, where $LFP(f)$ returns the least fixed-point of f. Our reduction states that the number of non-target users for each role set RS in the initial state can be limited to $|descendants(RS) \cap AR|$, which is an upper-bound on the number of administrative roles that can be assigned to users with role set RS. We state a lemma and then prove the correctness of this reduction.

Lemma 3.1 *Let* RS *be a role set. For every user* u *with the initial role set* RS*, the set of roles assigned to* u *is always a subset of* $descendants(RS)$*.*

Theorem 3.2 (Static involved-user reduction) *Let* $Q = \langle UA_0, U, u_t, \psi, goal\rangle$ *be a user-role reachability query,* UA_0' *be a set of user-role assignments obtained from* UA_0 *by reducing the number of users associated with each role set RS to* $|descendants(RS) \cap AR|$, $U' = \{u \mid (u,r) \in UA_0'\}$, *and* $Q' = \langle UA_0', U', u_t, \psi, goal\rangle$. *(a) If the goal is reachable in Q, then the goal is reachable in* Q'*; (b) If the goal is reachable in* Q'*, then the goal is reachable in Q.*

Proof sketch: Part (a) is proved based on the proof of Theorem 1 in [5] and Lemma 3.1. Let G and G' be the reduced state graph constructed for Q and Q', respectively. A user u is *involved* in a sequence of transitions T if u's role set changes in T. A user u is *essential* in a sequence of transitions T if there exists $s_i \overset{A_i}{\rightarrow} s_{i+1}$ in T such that u is the only user in s_{i+1} assigned the administrative role in A_i.

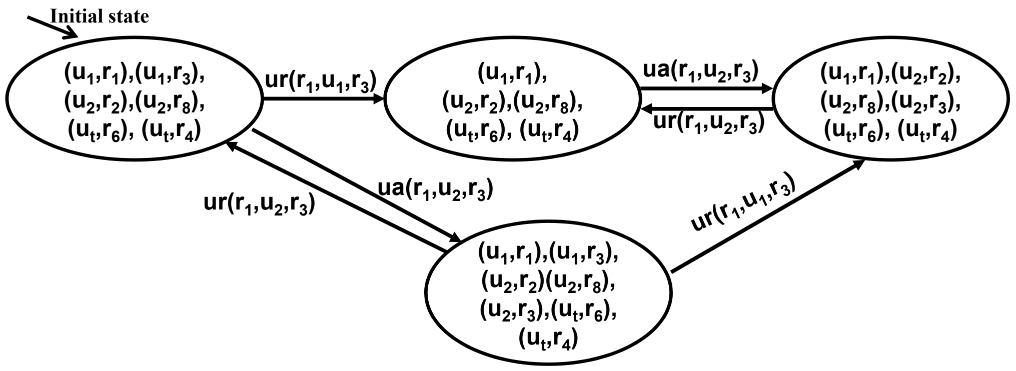
First, we show that, for every sequence of transitions T in G that contains a goal state, there exists a sequence of transitions T_1 in G that contains a goal state, and at most $|descendants(RS) \cap AR|$ non-target users in each role set *RS* are involved in T_1. T_1 is constructed as follows. If T contains at least one involved non-target user that is not essential, then we pick one such user u and remove from T all transitions that change u's role set. If all non-target users in T are essential, then we pick one such user u and remove all transitions that both change u's role set and are performed after the last state in which u is essential. The above process is repeated on the resulting sequence of transitions until no more transitions can be eliminated, which results in T_1. Because the above construction keeps in each state one user among those assigned an administrative role, the eliminated transitions do not disable transitions in T_1. Therefore, T_1 is a sequence of transitions in G. Because all users in T_1 are essential and for each user u with the initial role set RS, the set of roles assigned to u is always a subset of $descendants(RS)$ (Lemma 3.1), at most $|descendants(RS) \cap AR|$ non-target users with initial role set RS change their role sets in T_1. T_1 contains a goal state because the above construction does not eliminate transitions performed on the target user.

Next, we construct a sequence of transitions T' from T_1 such that T' is a sequence of transitions in G', and T' contains a goal state. We first remove all users that are not involved in T_1 from all states in T_1, which results in T_2. Let $T_2 = s_0 \overset{\alpha(r_{a1},u_1,r_1)}{\rightarrow} s_1 \ldots \overset{\alpha(r_{an},u_n,r_n)}{\rightarrow} s_n$ where $\alpha = ua/ur$. T' is constructed as $closure(UA'_0) \overset{\alpha(r_{a1},\sigma(u_1),r_1)}{\rightarrow} s'_1 \ldots \overset{\alpha(r_{an},\sigma(u_n),r_n)}{\rightarrow} s'_n$, where σ is a mapping between users in s_0 and users in $closure(UA'_0)$ defined below: $\sigma(u_i) = u_i$ if u_i is in $closure(UA'_0)$; otherwise, $\sigma(u_i) = u'_i$ where $\{r \mid (u_i, r) \in s_0\} = \{r \mid (u'_i, r) \in closure(UA'_0)\}$ and $u'_i \neq \sigma(u_j)$ for all $j \neq i$. Since T_2 contains at most $|descendants(RS) \cap AR|$ non-target users in each role set *RS* in UA_0, the above mapping is always possible. Therefore, if the goal is reachable in Q, then the goal is reachable in Q'.

Part (b) follows directly from the fact that UA'_0 is obtained from UA_0 by reducing the number of users associated with each role set RS to $|descendants(RS) \cap AR|$. □

The static involved-user reduction can be improved by not considering administrative roles in RS that are non-negative or irrevocable, i.e., limiting the number of non-target users with role set *RS* to $|(descendants(RS) \setminus \{r \mid r \in RS$ and r is non-negative or irrevocable$\}) \cap AR|$. Such roles will not be revoked during the analysis and hence do not need to be reassigned to non-target users.

Example 5 *Consider the user-role reachability query in Example 1. Our static involved-user reduction works as follows. The set of administrative roles in the ARBAC policy is $AR = \{r_1, r_2, r_6\}$. The non-target user u_1 is initially assigned role set*



Initial state
(u1,r1),(u1,r3), (u2,r2),(u2,r8), (ut,r6), (ut,r4)
ur(r1,u1,r3)
(u1,r1), (u2,r2),(u2,r8), (ut,r6), (ut,r4)
ua(r1,u2,r3)
ur(r1,u2,r3)
(u1,r1),(u2,r2), (u2,r8),(u2,r3), (ut,r6), (ut,r4)
ur(r1,u2,r3)
ua(r1,u2,r3)
ur(r1,u1,r3)
(u1,r1),(u1,r3), (u2,r2)(u2,r8), (u2,r3),(ut,r6), (ut,r4)

Figure 5: The state graph constructed in Example 5 with the static involved-user reduction.

$\{r_1, r_3\}$. *The policy does not contain a rule* $can\_assign(r_a, P \wedge \neg N, r)$ *such that* $P \subseteq \{r_1, r_3\}$*, so* $descendants(\{r_1, r_3\}) = \{r_1, r_3\}$ *and hence* $|descendants(\{r_1, r_3\}) \cap AR| = 1$. *This means that we need to keep* u_1 *in the initial state. Non-target users* u_2 *and* u_3 *are initially assigned role set* $\{r_2, r_8\}$. *Because the precondition of the first rule is a subset of* $\{r_2, r_8\}$*, the rule's target* r_3 *is a descendant. As a result,* $descendants(\{r_2, r_8\}) = \{r_2, r_3, r_8\}$ *and* $|descendants(\{r_2, r_8\}) \cap AR| = 1$. *This means that we need to keep either* u_2 *or* u_3*, but not both, in the initial state. Assume that we keep* u_2 *in the initial state. The state graph constructed with the static involved-user reduction is given in Figure 5.*

Spare-user reduction The spare-user reduction is similar to the static involved-user reduction, except that it limits the number of non-target users in every state, not only the initial state. Intuitively, the spare-user reduction should perform the same or better than the static involved-user reduction, since fewer or the same number of users are considered during the analysis. Our experimental results on a university RBAC and a university ARBAC policy show that this is true for most cases. However, in some cases, the spare-user reduction increases the numbers of states and transitions. For example, if a state s has been processed before a transition $s_1 \overset{A}{\to} s$, then removing spare users from s_1 may result in a different target state s' that, without the spare user reduction, might never be processed. In this case, the reduction could increase the size of the state graph.

Dynamic involved-user reduction The dynamic involved-user reduction limits the number of users whose role sets change in each explored path in the state graph. The algorithm performs a depth-first search (DFS) to check whether the goal is reachable. During the DFS, it keeps track of the number of users that change role sets (we call such users "involved users") in the run corresponding to the DFS stack. When the number of users involved in the run on the stack is greater than $|AR| + 1$, the algorithm explores only transitions which change role sets of users that are already involved in the trace.

Note that the above algorithm counts users that change role sets only in visible transitions. Informally, it would be unsound to also count users that change role sets in invisible transitions, because our algorithm always executes invisible transitions as soon as possible, but the invisible transitions do not necessarily appear in the paths in the full state graph (i.e., the graph without the invisible transition reduction) that reach the goal with the minimal number of involved users.

The dynamic involved-user reduction is partly incompatible with the user-count optimization in Section 3.3. With the user-count optimization, the algorithm does not keep track of identities of the users, which makes it hard to keep track of how many users changed role sets. One solution is to introduce identities for users on demand; we call this *the partial user-count reduction*. It works as follows. A state s is a set of tuples of the form $\langle Anon, Named, RS\rangle$, where *Anon* is the number of anonymous users with role set RS, and $Named$ is the set of named users with role set RS. The target user u_t is never anonymous. Thus, u_t has role set RS iff $u_t \in Named$. In every state, the algorithm computes transitions only for the named users and one anonymous user; the anonymous user becomes named in the target state of the transition. The name is chosen to be $\langle RS, i\rangle$, where i is the smallest i such that $\langle RS, i\rangle$ does not appear in $Named$.

On one hand, the dynamic involved-user reduction may generate smaller state graphs than the other two reductions presented in this section. On the other hand, tracking the involvement of users in the dynamic involved-user reduction imposes additional performance overhead and is partly incompatible with the user-count optimization. As a result, the performance of the dynamic involved-user reduction might or might not be better than other two reductions. In addition, the dynamic involved-user reduction is incompatible with the current implementation, which uses bread-first search. Consequently, we did not implement this reduction.

Example 6 *Consider a user-role reachability query which is the same as that in Example 1, except that all administrative roles in the ARBAC policy are replaced with r_1. Without any reduction, the state graph is the same as that in Figure 2. Because $AR = \{r_1\}$, $|AR| + 1 = 2$. Therefore, with the dynamic involved-user reduction, the number of users that change role sets in each path of the state graph is limited to 2. Figure 6 gives the state graph constructed with the dynamic involved-user reduction.*

Example 7 *As a small example of a query for which the dynamic involved-user reduction is more effective than the static involved-user reduction, consider a user-role reachability query which is the same as that in Example 1, except that all administrative roles in the ARBAC policy are replaced with r_1, u_2 is assigned only r_2 (instead of both r_2 and r_8), and the ARBAC policy contains one more rule $can\_assign(r_1, \{r_2\} \wedge \neg\{r_8\}, r_1)$. With no reduction, there are 8 states. The static involved user reduction does not eliminate any states because the number of users associated with each role set in the initial RBAC policy is 1 and for each role set RS, $|descendants(RS) \cap AR| = 1$. The dynamic involved user reduction reduces the number of states to 7.*

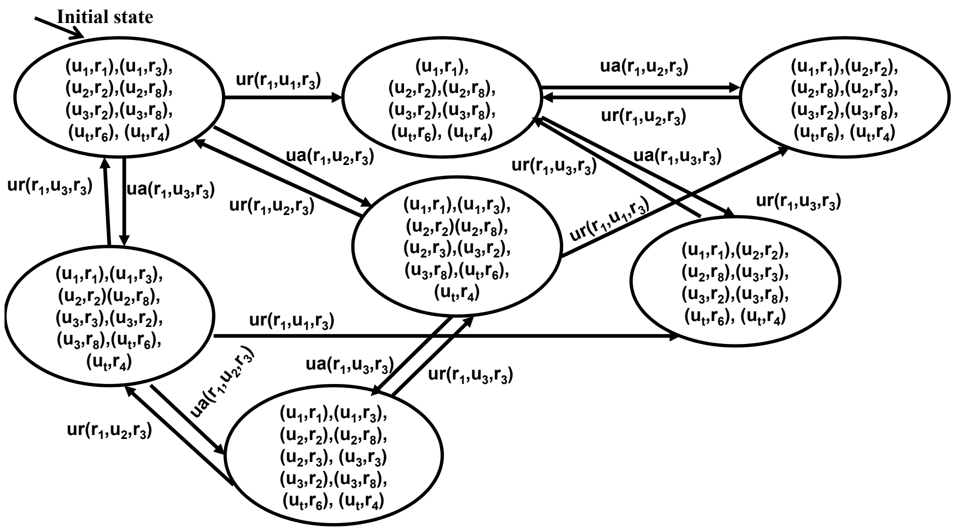

Initial state
(u1,r1),(u1,r3),(u2,r2),(u2,r8),(u3,r2),(u3,r8),(ut,r6),(ut,r4)
ur(r1,u1,r3)
(u1,r1),(u2,r2),(u2,r8),(u3,r2),(u3,r8),(ut,r6),(ut,r4)
ua(r1,u2,r3)
ur(r1,u2,r3)
(u1,r1),(u2,r2),(u2,r8),(u2,r3),(u3,r2),(u3,r8),(ut,r6),(ut,r4)
ua(r1,u2,r3)
ur(r1,u2,r3)
ur(r1,u3,r3)
ua(r1,u3,r3)
ur(r1,u3,r3)
ua(r1,u3,r3)
ur(r1,u3,r3)
(u1,r1),(u1,r3),(u2,r2)(u2,r8),(u2,r3),(u3,r2),(u3,r8),(ut,r6),(ut,r4)
ur(r1,u1,r3)
(u1,r1),(u2,r2),(u2,r8),(u3,r3),(u3,r2),(u3,r8),(ut,r6),(ut,r4)
(u1,r1),(u1,r3),(u2,r2)(u2,r8),(u3,r3),(u3,r2),(u3,r8),(ut,r6),(ut,r4)
ur(r1,u1,r3)
ua(r1,u2,r3)
ua(r1,u3,r3)
ur(r1,u3,r3)
ur(r1,u2,r3)
(u1,r1),(u1,r3),(u2,r2),(u2,r8),(u2,r3),(u3,r3)(u3,r2),(u3,r8),(ut,r6),(ut,r4)

Figure 6: The state graph constructed with the dynamic involved-user reduction.

3.5 Lazy Reduction

This section describes two lazy reduction techniques that reduce the size of the state graph by delaying transitions.

Subsumed-user reduction Let $roleset(u, s) = \{r \mid (u, r) \in s\}$. The subsumed-user reduction works as follows. Given a state s, for every non-target user u, if there exists a user u' other than u such that (1) $roleset(u, s) \subset roleset(u', s)$ and (2) $(roleset(u', s) \setminus roleset(u, s)) \cap Rel_{-} = \emptyset$ (i.e. all roles in $roleset(u', s) \setminus roleset(u, s)$ are non-negative), then we do not perform transitions on u from s. Such transitions will be performed later when one of the above conditions does not hold.

Intuitively, this reduction is sound because every transition enabled on u in s is also enabled on u' in s (since non-negative roles do not disable any transitions), and transitions performed on u' do not disable transitions enabled on u.

Given states s and s', we say that s' is a superstate of s, denoted as $s' \sqsupseteq s$, if $roleset(u_t, s') = roleset(u_t, s)$, and for every non-target user u in s, there exists a non-target user u' in s' such that $roleset(u, s) \subseteq roleset(u', s')$ and $(roleset(u', s') \setminus roleset(u, s)) \cap Rel_{-} = \emptyset$. For example, if r_3 is a non-negative role, then $\{(u, r_1), (u, r_3), (u', r_1), (u', r_2), (u', r_3)\} \sqsupseteq \{(u, r_1), (u, r_3), (u', r_1), (u', r_2)\}$.

Below, we prove one Lemma and then the correctness of the reduction.

Lemma 3 *Let $Q = \langle UA_0, U, u_t, \psi, goal\rangle$ be a user-role reachability query, $s_0 = closure(UA_0)$, and G_{sub} and G be reduced state graphs constructed for Q with and without the subsumed-user reduction, respectively. (a) For every sequence of transitions $s_0 \overset{A_1}{\rightarrow} s_1 \overset{A_2}{\rightarrow} \ldots \overset{A_n}{\rightarrow} s_n$ in G_{sub}, there exists a sequence of transitions $s_0 \overset{A_1}{\rightarrow} s_1 \overset{A_2}{\rightarrow} \ldots \overset{A_n}{\rightarrow} s_n$ in G. (b) For every sequence of transitions $s_0 \overset{A_1}{\rightarrow} s_1 \overset{A_2}{\rightarrow} \ldots \overset{A_n}{\rightarrow} s_n$ in G, there exists a sequence of transitions $s_0 \overset{A'_1}{\rightarrow} s'_1 \overset{A'_2}{\rightarrow} \ldots \overset{A'_m}{\rightarrow} s'_m$ in G_{sub}, such that $s'_m \sqsupseteq s_n$.*

Proof: Part (a) follows from the fact that, in every state, the algorithm with the subsumed user reduction explores a subset of the enabled transitions.

The proof of part (b) is by induction on n.

Base case: When $n = 0$, both G and G_{sub} contain only state s_0. Therefore, the lemma holds.

Induction hypothesis: Assume that the lemma holds for $n = k$, i.e., for every sequence of transitions $s_0 \overset{A_1}{\rightarrow} s_1 \overset{A_2}{\rightarrow} \ldots \overset{A_k}{\rightarrow} s_k$ in G, there exists a sequence of transitions $s_0 \overset{A'_1}{\rightarrow} s'_1 \overset{A'_2}{\rightarrow} \ldots \overset{A'_m}{\rightarrow} s'_m$ in G_{sub} such that $s'_m \sqsupseteq s_k$. We now prove the lemma for $n = k+1$. Assume that there is a transition $s_k \overset{ua(r_a,u,r)}{\rightarrow} s_{k+1}$ in G. If $u = u_t$, then because $s'_m \sqsupseteq s_k$, $roleset(u_t, s_k) = roleset(u_t, s'_m)$ and s'_m contains all administrative roles in s_k. Therefore, G_{sub} contains a transition $s'_m \overset{ua(r_a,u_t,r)}{\rightarrow} s'_{m+1}$ such that $s'_{m+1} \sqsupseteq s_{k+1}$. Otherwise, since for every non-target user u in s_k, there exists a non-target user u' in s'_m such that $roleset(u, s_k) \subseteq roleset(u', s'_m)$ and $(roleset(u', s'_m) \backslash roleset(u, s_k)) \cap Rel_- = \emptyset$, G_{sub} contains a transition $s'_m \overset{ua(r_a,u',r)}{\rightarrow} s'_{m+1}$ such that $s'_{m+1} \sqsupseteq s_{k+1}$. The case where the transition is a ur transition can be similarly proved. Therefore, part (b) holds. □

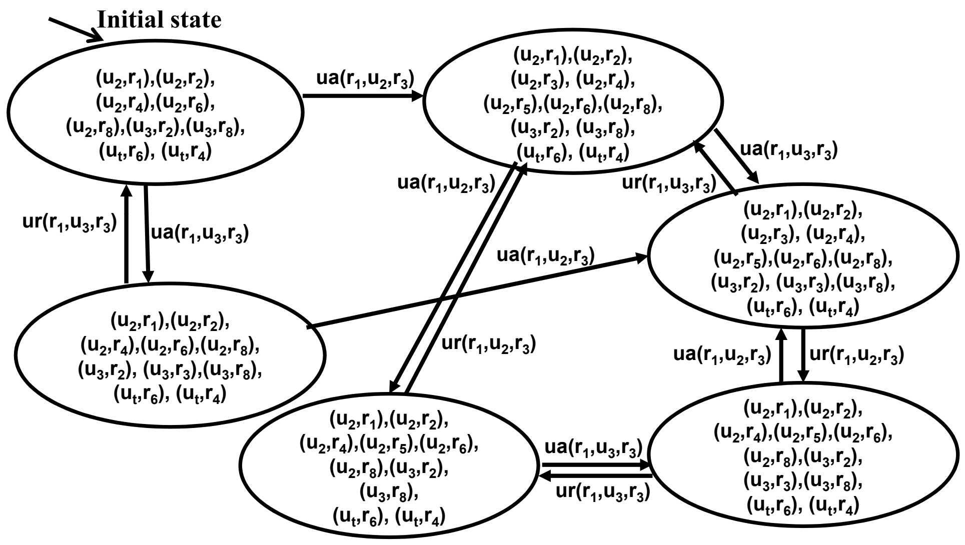
Theorem 4 *Let $Q = \langle UA_0, U, u_t, \psi, goal\rangle$ be a user-role reachability query, $s_0 = closure(UA_0)$, and G_{sub} and G be reduced state graphs constructed for Q with and without the subsumed-user reduction, respectively. The goal is reachable in G iff the goal is reachable in G_{sub}.*

Proof: The "if" part follows immediately from part (a) of Lemma 3.

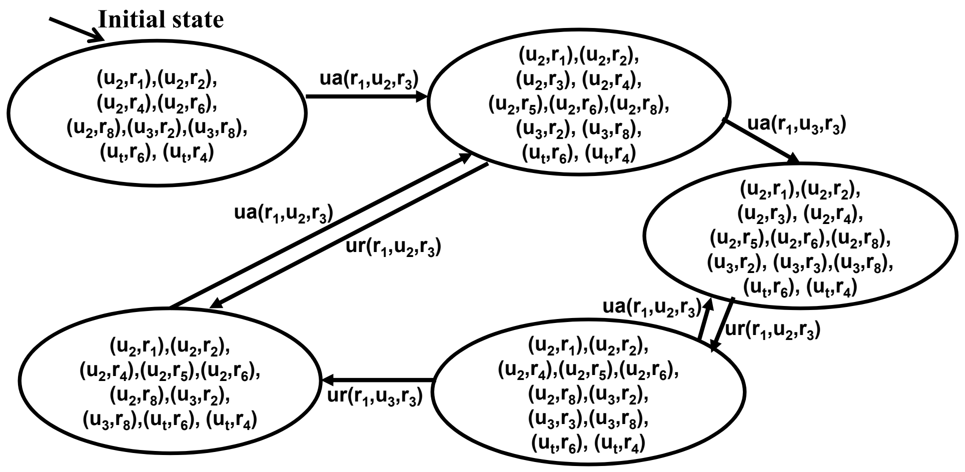
Next, we prove the "only if" part. Suppose G contains a sequence of transitions $s_0 \overset{A_1}{\rightarrow} s_1 \overset{A_2}{\rightarrow} \ldots \overset{A_n}{\rightarrow} s_g$, such that $goal \subseteq roleset(u_t, s_g)$. From part (b) of Lemma 3, G_{sub} contains a sequence of transitions $s_0 \overset{A_1}{\rightarrow} s'_1 \overset{A'_2}{\rightarrow} \ldots \overset{A'_m}{\rightarrow} s'_m$, such that $s'_m \sqsupseteq s_g$. The latter implies $roleset(u_t, s_g) = roleset(u_t, s'_m)$ and hence $goal \subseteq roleset(u_t, s'_m)$. □

Example 8 *Consider Example 1, except with the initial RBAC policy $UA_0 = \{(u_2, r_1)$, (u_2, r_2), (u_2, r_8), (u_3, r_2), (u_3, r_8), $(u_t, r_6)\}$. Figures 7(a) and 7(b) give the state graph computed without and with the subsumed-user reduction, respectively. In the initial state, the role sets of u_2 and u_3 are $\{r_1, r_2, r_4, r_6, r_8\}$ and $\{r_2, r_8\}$, respectively, and r_1, r_4, and r_6 are non-negative roles. Therefore, with the subsumed-user reduction, the algorithm does not compute transitions for u_3 from the initial state. Similarly, $ur(r_1, u_3, r_3)$ is not performed from state $\{(u_2, r_1)$, (u_2, r_2), (u_2, r_3), (u_2, r_4), (u_2, r_6), (u_2, r_8), (u_3, r_2), (u_3, r_3), (u_3, r_8), (u_t, r_4), $(u_t, r_6)\}$ and $ua(r_1, u_3, r_3)$ is not performed from state $\{(u_2, r_1)$, (u_2, r_2), (u_2, r_3), (u_2, r_4), (u_2, r_6), (u_2, r_8), (u_3, r_2), (u_3, r_8), (u_t, r_4), $(u_t, r_6)\}$.*

Delayed revocation A ur transition can be delayed if the transition can neither enable new transitions in s nor be disabled by any transitions. Formally, a transition $s \overset{ur(r_a,u,r)}{\rightarrow} s'$ is not performed from s (i.e. is delayed) if


Initial state
(u2,r1),(u2,r2),(u2,r4),(u2,r6),(u2,r8),(u3,r2),(u3,r8),(ut,r6), (ut,r4)
ua(r1,u2,r3)
(u2,r1),(u2,r2),(u2,r3), (u2,r4),(u2,r5),(u2,r6),(u2,r8),(u3,r2), (u3,r8),(ut,r6), (ut,r4)
ua(r1,u3,r3)
ur(r1,u3,r3)
ur(r1,u3,r3)
ua(r1,u3,r3)
ua(r1,u2,r3)
(u2,r1),(u2,r2),(u2,r3), (u2,r4),(u2,r5),(u2,r6),(u2,r8),(u3,r2), (u3,r3),(u3,r8),(ut,r6), (ut,r4)
ua(r1,u2,r3)
(u2,r1),(u2,r2),(u2,r4),(u2,r6),(u2,r8),(u3,r2), (u3,r3),(u3,r8),(ut,r6), (ut,r4)
ur(r1,u2,r3)
ua(r1,u2,r3)
ur(r1,u2,r3)
(u2,r1),(u2,r2),(u2,r4),(u2,r5),(u2,r6),(u2,r8),(u3,r2),(u3,r8),(ut,r6), (ut,r4)
ua(r1,u3,r3)
ur(r1,u3,r3)
(u2,r1),(u2,r2),(u2,r4),(u2,r5),(u2,r6),(u2,r8),(u3,r2),(u3,r3),(u3,r8),(ut,r6), (ut,r4)

(a)


Initial state
(u2,r1),(u2,r2),(u2,r4),(u2,r6),(u2,r8),(u3,r2),(u3,r8),(ut,r6), (ut,r4)
ua(r1,u2,r3)
(u2,r1),(u2,r2),(u2,r3), (u2,r4),(u2,r5),(u2,r6),(u2,r8),(u3,r2), (u3,r8),(ut,r6), (ut,r4)
ua(r1,u3,r3)
ua(r1,u2,r3)
ur(r1,u2,r3)
(u2,r1),(u2,r2),(u2,r3), (u2,r4),(u2,r5),(u2,r6),(u2,r8),(u3,r2), (u3,r3),(u3,r8),(ut,r6), (ut,r4)
ua(r1,u2,r3)
ur(r1,u2,r3)
(u2,r1),(u2,r2),(u2,r4),(u2,r5),(u2,r6),(u2,r8),(u3,r2),(u3,r8),(ut,r6), (ut,r4)
ur(r1,u3,r3)
(u2,r1),(u2,r2),(u2,r4),(u2,r5),(u2,r6),(u2,r8),(u3,r2),(u3,r3),(u3,r8),(ut,r6), (ut,r4)

(b)

Figure 7: The state graph constructed in Example 8: (a) without the subsumed-user reduction, (b) with the subsumed-user reduction.

1. $trans(s) \supseteq trans(s') \cup \{ur(r_a, u, r)\}$, where $trans(s)$ and $trans(s')$ are sets of all visible transitions enabled from s and s', respectively,

2. $s' \setminus s = \emptyset$,

3. $trans(s)$ contains at least one ua transition, and

4. r_a is non-negative or irrevocable.

Conditions 1 and 2 specify that $s \overset{ur(r_a,u,r)}{\rightarrow} s'$ does not enable new visible and invisible transitions, respectively. Conditions 3 and 4 specify that $s \overset{ur(r_a,u,r)}{\rightarrow} s'$ cannot be disabled by other transitions. Requiring $trans(s)$ to contain at least one ua transition ensures that not all transitions from s will be delayed, which in turn ensures that the delayed ur transitions can be executed later. This condition can be relaxed by not delaying one ur transition that is enabled in s, if no ua transitions are enabled in s.

The correctness of the delayed revocation reduction is formalized in Theorem 6. We prove one lemma and then Theorem 6. Let $\sqsubseteq$ denote the subsequence notation.

Lemma 5 *Let $Q = \langle UA_0, U, u_t, \psi, goal\rangle$ be a user-role reachability query, $s_0 = closure(UA_0)$, and G_{dr} and G be reduced state graphs constructed for Q with and without the delayed revocation reduction, respectively. (a) For every sequence of transitions $s_0 \overset{A_1}{\rightarrow} s_1 \overset{A_2}{\rightarrow} \dots \overset{A_n}{\rightarrow} s_n$ in G, there exists a sequence of transitions $s_0 \overset{A'_1}{\rightarrow} s'_1 \overset{A'_2}{\rightarrow} \dots \overset{A'_m}{\rightarrow} s'_m$ in G_{dr} such that $\langle A'_1, \dots, A'_m\rangle \sqsubseteq \langle A_1, \dots, A_n\rangle$, and $s'_m = s_n \cup \{(u, r) \mid ur(\_, u, r) \in \langle A_1, \dots, A_n\rangle \setminus \langle A'_1, \dots, A'_m\rangle\}$. (b) For every sequence of transitions $s_0 \overset{A_1}{\rightarrow} s_1 \overset{A_2}{\rightarrow} \dots \overset{A_n}{\rightarrow} s_n$ in G_{dr}, there exists a sequence of transitions $s_0 \overset{A_1}{\rightarrow} s_1 \overset{A_2}{\rightarrow} \dots \overset{A_n}{\rightarrow} s_n$ in G.*

PROOF: Part (a) is proved by induction on the length n.

Base Case: If n = 0, then both G and G_{dr} comprise only one state s_0. The lemma holds.

Induction: Assume the lemma holds for $n = k$. Below, we prove the lemma for $n = k + 1$. Suppose there is a sequence of transitions $s_0 \overset{A_1}{\rightarrow} s_1 \overset{A_2}{\rightarrow} \dots \overset{A_{k+1}}{\rightarrow} s_{k+1}$ in G. By the induction hypothesis, there is a sequence of transitions $s_0 \overset{A'_1}{\rightarrow} s'_1 \overset{A'_2}{\rightarrow} \dots \overset{A'_j}{\rightarrow} s'_j$ in G_{dr} such that $\langle A'_1, \dots, A'_j\rangle \sqsubseteq \langle A_1, \dots, A_k\rangle$, and $s'_j = s_k \cup \{(u, r) \mid ur(\_, u, r) \in \langle A_1, \dots, A_k\rangle \setminus \langle A'_1, \dots, A'_j\rangle\}$.

First, we consider the case where A_{k+1} is not a delayed ur transition in s_k. Since at least one ua transition is enabled from s and r_a is non-negative or irrevocable, delayed ur transitions do not disable A_{k+1}. In addition, s_k differs from s'_j only by containing user-role pairs removed from s_k by delayed ur transitions. As a result, transition $s'_j \overset{A_{k+1}}{\rightarrow} s'_{j+1}$ can be executed where $s'_{j+1} = s_{k+1} \cup \{(u, r) \mid ur(\_, u, r) \in \langle A_1, \dots, A_k, A_{k+1}\rangle \setminus \langle A'_1, \dots, A'_j, A_{k+1}\rangle\}$. If A_{k+1} is a delayed ur transition in s_k, then $s'_j = s_{k+1} \cup \{(u, r) \mid ur(\_, u, r) \in \langle A_1, \dots, A_k, A_{k+1}\rangle \setminus \langle A'_1, \dots, A'_j\rangle\}$.

Part (b) follows from the fact that G_{dr} is a subgraph of G. Thus, the lemma holds.

Theorem 6 *Let $Q = \langle UA_0, U, u_t, \psi, goal\rangle$ be a user-role reachability query, $s_0 = closure(UA_0)$, and G_{dr} and G be reduced state graphs constructed for Q with and without the delayed revocation reduction, respectively. The goal is reachable in G iff the goal is reachable in G_{dr}.*

PROOF: If the goal is reachable in G, then there exists a sequence of transitions $s_0 \overset{A_1}{\rightarrow} s_1 \overset{A_2}{\rightarrow} \dots \overset{A_n}{\rightarrow} s_g$ in G such that $goal \subseteq roleset(u_t, s_g)$. From Lemma 5(a), there exists a sequence of transitions $s_0 \overset{A'_1}{\rightarrow} s'_1 \overset{A'_2}{\rightarrow} \dots \overset{A'_m}{\rightarrow} s'_m$ in G_{dr} such that $s_g \subseteq s'_m$, so $goal \subseteq roleset(u_t, s'_m)$. Thus, the goal is reachable in G_{dr}.

If G_{dr} contains state s_g such that $goal \subseteq s_g$, then from Lemma 5(b), s_g is also in G and hence the goal is reachable. Therefore, the theorem holds.

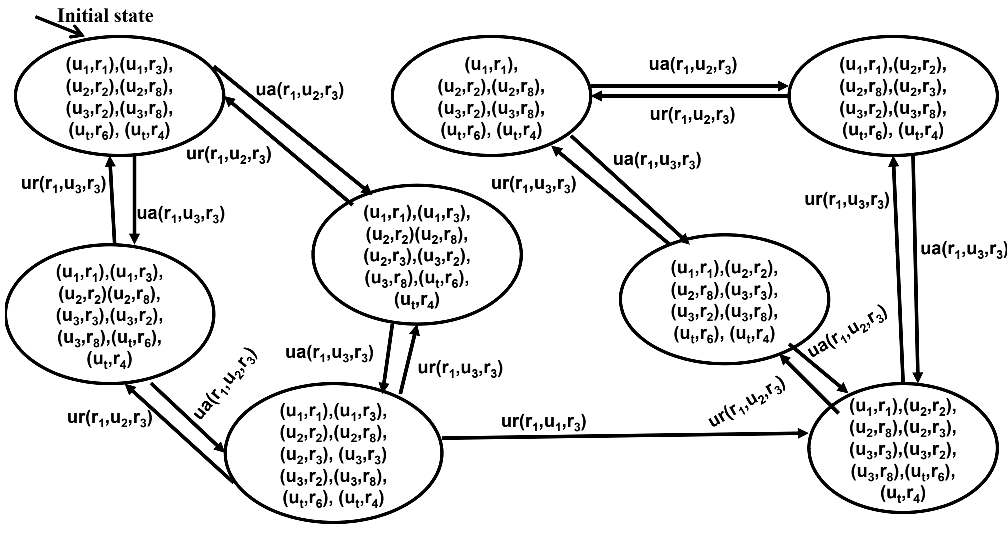

Initial state
(u1,r1),(u1,r3), (u2,r2),(u2,r8), (u3,r2),(u3,r8), (ut,r6), (ut,r4)
ua(r1,u2,r3)
ur(r1,u2,r3)
ur(r1,u3,r3)
ua(r1,u3,r3)
(u1,r1),(u1,r3), (u2,r2)(u2,r8), (u3,r3),(u3,r2), (u3,r8),(ut,r6), (ut,r4)
(u1,r1),(u1,r3), (u2,r2)(u2,r8), (u2,r3),(u3,r2), (u3,r8),(ut,r6), (ut,r4)
ua(r1,u3,r3)
ur(r1,u3,r3)
ua(r1,u2,r3)
ur(r1,u2,r3)
(u1,r1),(u1,r3), (u2,r2),(u2,r8), (u2,r3), (u3,r3) (u3,r2),(u3,r8), (ut,r6), (ut,r4)
ur(r1,u1,r3)
(u1,r1), (u2,r2),(u2,r8), (u3,r2),(u3,r8), (ut,r6), (ut,r4)
ua(r1,u2,r3)
ur(r1,u2,r3)
(u1,r1),(u2,r2), (u2,r8),(u2,r3), (u3,r2),(u3,r8), (ut,r6), (ut,r4)
ua(r1,u3,r3)
ur(r1,u3,r3)
(u1,r1),(u2,r2), (u2,r8),(u3,r3), (u3,r2),(u3,r8), (ut,r6), (ut,r4)
ur(r1,u3,r3)
ua(r1,u3,r3)
ua(r1,u2,r3)
ur(r1,u2,r3)
(u1,r1),(u2,r2), (u2,r8),(u2,r3), (u3,r3),(u3,r2), (u3,r8),(ut,r6), (ut,r4)

Figure 8: The state graph constructed in Example 9 with the delayed revocation reduction.

Example 9 *Consider the user-role reachability query in Example 1. Since the $ur(r_1, u_1, r_3)$ transition does not enable new transitions from the initial state and r_1 is non-negative, with delayed revocation reduction, this transition is not performed from the initial state. Similarly, $ur(r_1, u_1, r_3)$ is also not performed from state $\{(u_1, r_1)$, (u_1, r_3), (u_2, r_2), (u_2, r_8), (u_3, r_2), (u_3, r_3), (u_3, r_8), (u_t, r_6), $(u_t, r_4)\}$ and state $\{(u_1, r_1)$, (u_1, r_3), (u_2, r_2), (u_2, r_3), (u_2, r_8), (u_3, r_2), (u_3, r_8), (u_t, r_6), $(u_t, r_4)\}$. The graph constructed with the delayed revocation reduction is given in Figure 8.*

3.6 Hierarchical Rule Reduction

Hierarchical rule reduction avoids considering rules whose administrative roles are junior to non-negative or irrevocable administrative roles in UA_0. Formally, the hierarchical rule reduction eliminates rules $can\_assign(r_a, \_, \_)$ such that there exists $(\_, r') \in UA_0$ such that $r' \succeq r_a$ and r' is non-negative or irrevocable. This is safe because senior roles inherit all administrative permissions of their junior roles, and non-negative/irrevocable roles are never revoked during analysis. Note that, the hierarchical rule reduction does not reduce the size of the transition graph, but may reduce the analysis time, since fewer rules are applied during analysis.

Consider the user-role reachability analysis query in Example 1 with the role hierarchy $r_1 \succeq r_2$. The following three rules are added when the policy is transformed into a non-hierarchical policy: $can\_assign(r_1, \{r_8, r_1\} \wedge \neg\emptyset, r6)$, $can\_assign(r_1, \{r_6\} \wedge \neg\emptyset, r_7)$, and $can\_assign(r_1, \{r_1\} \wedge \neg\emptyset, r_3)$. Since r_1 is a non-negative role, r_1 will never be revoked during analysis. As a result, rules 4 and 5 in Example 1 are not necessary for reaching the goal (since administrative roles of these two rules are r_2, which is junior to r_1), and hence will not be applied during analysis with this reduction.

Algorithm 4 Integrating all reductions.

```
1: procedure $allreduct(UA_0, U, u_t, \psi, goal)$
2: $\psi_1 = deadrole(UA_0, \psi, goal)$;
3: $(tarRel_+, tarRel_-, \psi_t) = targetslicing(UA_0, u_t, \psi_1, goal)$;
4: $(ntarRel_+, ntarRel_-, \psi_{nt}) = nontarslicing(UA_0, u_t, \psi_1, goal)$;
5: $init = \{closure_1(U, u_t, UA_0, tarRel_+, tarRel_-, \psi_t, ntarRel_+, ntarRel_-, \psi_{nt})\}$;
6: $(tarRel1_+, tarRel1_-, \psi1_t) = targetslicing(init, u_t, \psi_1, goal)$;
7: $(ntarRel1_+, ntarRel1_-, \psi1_{nt}) = nontarslicing(init, u_t, \psi_1, goal)$;
8: $(\psi2_t, \psi2_{nt}) = hierarchicalRuleReduction(init, \psi1_t, \psi1_{nt})$;
9: $init' = staticInvolvedUserReduction(init, u_t, \psi2_t, \psi2_{nt})$;
10: $W = \{init'\}$;
11: while $W \neq \emptyset$ do
12:     remove a state $s$ from $W$;
13:     for each $(Uset, Rset) \in s$ do
14:         if $subsume(s, u_t, Uset, Rset) \neq true$ then
15:             $W = W \cup userEquivReduction(s, u_t, \psi2_t, \psi2_{nt}, goal)$;
16:         end if
17:     end for
18: end while
```

3.7 Integrating All Reductions

Integrating all reductions needs to be done carefully to achieve the best performance, since the order of the reductions may significantly affect the performance of the algorithm. Let $Q = \langle UA_0, U, u_t, \psi, goal\rangle$ be a user-role reachability query. Algorithm 4 describes the integration process.

1. Perform the dead role analysis to eliminate ARBAC rules involving dead roles (line 2) and store the rest of rules in ψ_1.

2. Perform the enhanced slicing (lines 3–4) on ψ_1, and store the sliced policies for the target user and non-target users in ψ_t and ψ_{nt}, respectively. The order of steps 1 and 2 is not significant.

3. Compute the initial state $init$ from the initial RBAC policy UA_0; invisible transitions for the target user are computed using rules in ψ_t and invisible transitions for non-target users are computed using rules in ψ_{nt} (line 5). Note: this step is performed after steps 1 and 2, because steps 1 and 2 reduce the number of rules applied to compute the initial state and hence may reduce the size of the initial state.

4. Perform the second-stage slicing on ψ_1 and $init$, and store the sliced policies for the target user and non-target users in $\psi1_t$ and $\psi1_{nt}$, respectively (lines 6–7). Note: this step is performed after step 3 because it depends on $init$ computed in step 3.

5. Perform the hierarchical rule reduction on $\psi1_t$ and $\psi1_{nt}$, and store the resulting policies in $\psi2_t$ and $\psi2_{nt}$ (line 8). Note: this step is performed after steps 1–4,

because a negative administrative role in the original policy may become non-negative after dead role analysis and two-stage slicing, which may improve the effectiveness of the hierarchical rule reduction.

6. Perform the static involved user reduction (line 9) to keep in the system the target user and at most $|AR|$ non-target users for each role set, where AR is the set of administrative roles that may possibly be assigned to non-target users during analysis, computed as the intersection of administrative roles in $\psi2_t$ and target roles in $\psi2_{nt}$. Note: step 6 is performed after steps 1–5, because steps 1–5 may reduce the number of rules considered and hence may reduce the number of roles in AR.

7. During the analysis, for every state s and every user equivalent set $(\mathit{Uset}, \mathit{Rset})$ in s, apply the subsumed-user reduction to check whether s contains a user equivalent set $(\mathit{Uset}', \mathit{Rset}')$ such that $\mathit{Rset} \subseteq \mathit{Rset}'$ and $\mathit{Rset}' \setminus \mathit{Rset}$ contains only non-negative roles (function $\mathit{subsume}$). If so, do not explore transitions for users in Uset; otherwise, explore transitions from s for users in $(\mathit{Uset}, \mathit{Rset})$ using the user equivalent set reduction (lines 10–18).

Note that, even if the enhanced slicing alone reduces the number of states and transitions computed, integrating the enhanced slicing and the user-equivalent set reduction does not always reduce the number of states and transitions. This is because, sometimes, when fewer roles are assigned to users, the state may contain more user equivalent sets. In our experiments, the integration reduces the number of states and transitions for most policies, but increases the number of states and transitions for a few policies.

4 Parallel Version of User-Role Reachability Analysis Algorithm in [22]

Multi-core processors are becoming pervasive. To benefit from this, it is important to parallelize algorithms. This section modifies the algorithm from [22] (Algorithm 1 in this paper) to perform analysis in parallel. The basic idea is to create multiple threads, each of which repeatedly removes and processes states from the workset. Since more than one thread may access the workset and the set of processed states concurrently, we explored techniques to efficiently synchronize and reduce contention on these two sets. Pseudocode for the parallel algorithm is given in Algorithm 5.

First, we perform slicing to eliminate irrelevant roles, as we do in Algorithm 1. We then compute the initial state init of the transition graph and add init to a workset W (Line 5). Next, we create n threads $t_0, \ldots, t_n$ (Line 6; $||$ represents the concurrent execution of threads). Typically, n is greater than the number of cores in order to keep all cores busy. Finally, each thread t_i removes one state from the workset W, computes transitions enabled from the state using Lines 7 –10 and 14–17 of Algorithm 1, and adds the target states to W and the set of reachable states $\mathit{Reached}$ if the target

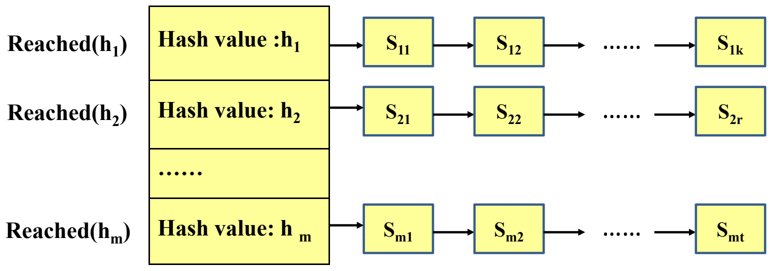


Figure 9: Implementation of the set of reachable states *Reached*.

Algorithm 5 User-Role Reachability Analysis Algorithm in [22].

```
1:  $Reached = W = Rel_+ = Rel_- = \emptyset$; $RelRule = \emptyset$; $done = 0$;
2:  procedure $mcanalysis(UA_0, U, u_t, \psi, goal)$
3:      $(Rel_+, Rel_-, RelRule) = slicing(UA_0, \psi, goal)$; $init = closure(U, u_t, UA_0)$;
4:      if $goal \subseteq \{r \mid (u_t, r) \in init\}$ then return true; end if
5:      $W = Reached(h(init)) = \{init\}$;
6:      $start(t_1) \mid\mid \ldots \mid\mid start(t_n)$;
7:  procedure $start(t_i)$
8:  while !done do
9:      if( $W == \emptyset$ and all threads are idle) then done = 1; end if
10:     while ($W \neq \emptyset$)
11:         $lock(W)$; remove a state $s$ from $W$;   $unlock(W)$;
12:         for all transitions $s \overset{ua(r_a, u, r)}{\to} s'$
13:             if $goal \subseteq \{r \mid (u_t, r) \in s'\}$ then return true; end if
14:             $lock(Reached(h(s')))$;
15:             if ($Reached(h(s'))$ does not exist)
16:                 $Reached(h(s')) = \{s'\}$; $unlock(Reached(h(s')))$;
17:                 $lock(W)$; $W = W \cup \{s'\}$; $unlock(W)$;
18:             else if ($s' \notin Reached(h(s'))$)
19:                 $Reached(h(s')) = Reached(h(s')) \cup \{s'\}$; $unlock(Reached(h(s')))$;
20:                 $lock(W)$; $W = W \cup \{s'\}$; $unlock(W)$;
21:                 else $unlock(Reached(h(s')))$; end if
22:     end if    end for    end while
23: end while
24: return false;
```

states are not already in *Reached* (Lines 11–21). Since multiple threads may access *Reached* concurrently, synchronization is needed to prevent interference between concurrent accesses. To reduce contention on *Reached*, we implemented *Reached* as a hashtable with a separate lock for each bucket (i.e. each collision chain), shown in Figure 9. As usual, $Reached(h)$ stores a set of states whose hash values are h. After a thread computes a transition $s \xrightarrow{\alpha} s'$, it computes the hash value $h(s')$ of s', locks $Reached(h(s'))$, adds s' to $Reached(h(s'))$ if s' is not already in $Reached(h(s'))$, and unlocks $Reached(h(s'))$. Our experimental results show that locking each bucket instead of the entire hashtable significantly improves the performance, because threads access $Reached$ very frequently and checking whether a state is in $Reached$ is relatively expensive. The algorithm terminates if the goal is reached, or if W is empty and all threads are not performing any computation.

To reduce contention on the workset W, we modify the algorithm so that each thread has its own workset. Below, we present three approaches to synchronization for the workset.

- **NoLock:** In this approach, a thread never accesses other threads' worksets. Every time a thread computes a transition, it stores the target state in its own workset, if the target state is not already in $Reached$. This approach eliminates all locking for worksets, but may result in idle threads (due to empty worksets).

- **FullLock:** In this approach, a thread t is allowed to access other threads' worksets to remove a state to process, if t's workset is empty. The thread t will access other threads' worksets one by one until it finds a non-empty workset, when it removes one or more states from the workset. This approach provides better load balancing, but it requires locking the workset every time it is accessed. A challenge is to decide how many states a thread should remove from another thread's workset at a time. If a thread takes one state at a time, then the thread is likely to run out of work again relatively soon. Another approach is to take a fixed fraction of the states in the workset. In our implementation, we pick $max(0.5 * sizeof(workset), maxTake)$ states, where $sizeof(workset)$ is the size of the workset and $maxTake$ is a user-specified bound.

- **PartialLock:** In this approach, after a thread t_i computes a fixed number of transitions, it checks if thread $t_{(i-1)\ mod\ n}$ is idle. If so, it locks the workset of thread $t_{(i-1)\ mod\ n}$, adds the target state to the workset, unlocks the workset, and starts thread $t_{(i-1)\ mod\ n}$. The advantage of this approach is that locking is needed only when thread t_i adds a state to thread $t_{(i-1)\ mod\ n}$'s workset. A challenge is to decide how many transitions a thread should compute before it checks if a thread is idle. If the number is small, then each thread t_i has to frequently check if $t_{(i-1)modn}$ is idle. If the number is large, then a thread may be idle for long time.

Other designs: It may be possible to improve performance by replacing mutual exclusion locks on buckets in $Reached$ with *reader-writer locks*. However, our experiments show that this does not yield performance improvement. In fact, it often causes performance degradation, because multiple threads rarely access the same bucket simultaneously, and reader-writer locks, due to their complexity, incur greater overhead than mutual exclusion locks.

We also considered using a lock-free data structure to implement the workset. To evaluate the potential benefit of this approach, we measured the overhead of the mutual exclusion lock by comparing running times of one thread with and without locking. Our experimental results show that locking incurs negligible overhead. This implies that a lock-free data structure would not appreciably improve the performance.

5 Performance Results

This section evaluates the effectiveness of our reduction techniques and parallel algorithms using the university ARBAC policy developed in [22] and the university RBAC policy developed in [7].

The university RBAC and ARBAC policies contain 845 users, 32 roles (including 13 administrative roles), 329 can_assign rules, and 78 can_revoke rules, after being converted to the corresponding non-hierarchical policies. The policies include rules for assignment of users to various student and employee roles. Student roles include undergraduate student, graduate student, teaching assistant, research assistant, honors student, etc. Employee roles include president, provost, dean, department chair, faculty, honors program director, etc. A sample can_assign rule is: *the honors program director can assign an undergraduate student to the honors student role.* A sample user-role reachability query is: *can a user who is a member of the department chair role and a user who is a member of the undergraduate student role assign the latter user to the honors student role?* Details of the university ARBAC policy are available from [21].

The university ARBAC policy does not satisfy the separate administration restriction. In addition, the policy has hierarchical role assignment w.r.t all administrative roles except those for assigning users to roles "honors student" and "graduate student". This means that if the goal contains these two roles, then we cannot directly apply the algorithm for analyzing ARBAC with separate administration to carry out analysis. In our experiments, we randomly select one target user u_t, one role r, and n non-target users $\{u_1, \ldots, u_n\}$. We then apply analysis algorithms to check whether users in $\{u_1, \ldots, u_n, u_t\}$ together can assign u_t to both honors student role and role r.

Each data point reported in this section is an average over 8 randomly generated queries. These 8 queries were generated as follows. First, we randomly generated 100 queries. Next, we ran the analysis program without reduction for the university policy containing 50 users and chose the first 8 policies in which the program generated at least 30 states. This eliminates very easy queries. The sets of non-target users were generated incrementally, so that each set of non-target users is a superset of the sets of non-target users generated for smaller n.

Effectiveness of reduction techniques Figures 10(a)–(d) report the number of states, the number of transitions, the execution time, and the number of timeout policies (i.e., policies for which the analysis does not complete within 5 hours) for the program without reduction (NoReduct) and with each of the four least effective reductions: the dead-role reduction (DeadRole), the delayed revocation reduction (DelayedRev), the subsumed-user reduction (Subsumed), and the static involved-user reduction (Involved). Figure 11 reports the same for NoReduct and the five most effective reductions: the enhanced slicing (EnhancedSlicing), the user equivalent set reduction (UserEquiv), the spare-user reduction (Spare), all reductions in [24] (AllReduct in [24]) and all reductions in this paper (AllReduct). The reported data were obtained on a 2.5GHz Pentium machine with 4GB RAM running Ubuntu 12.04.

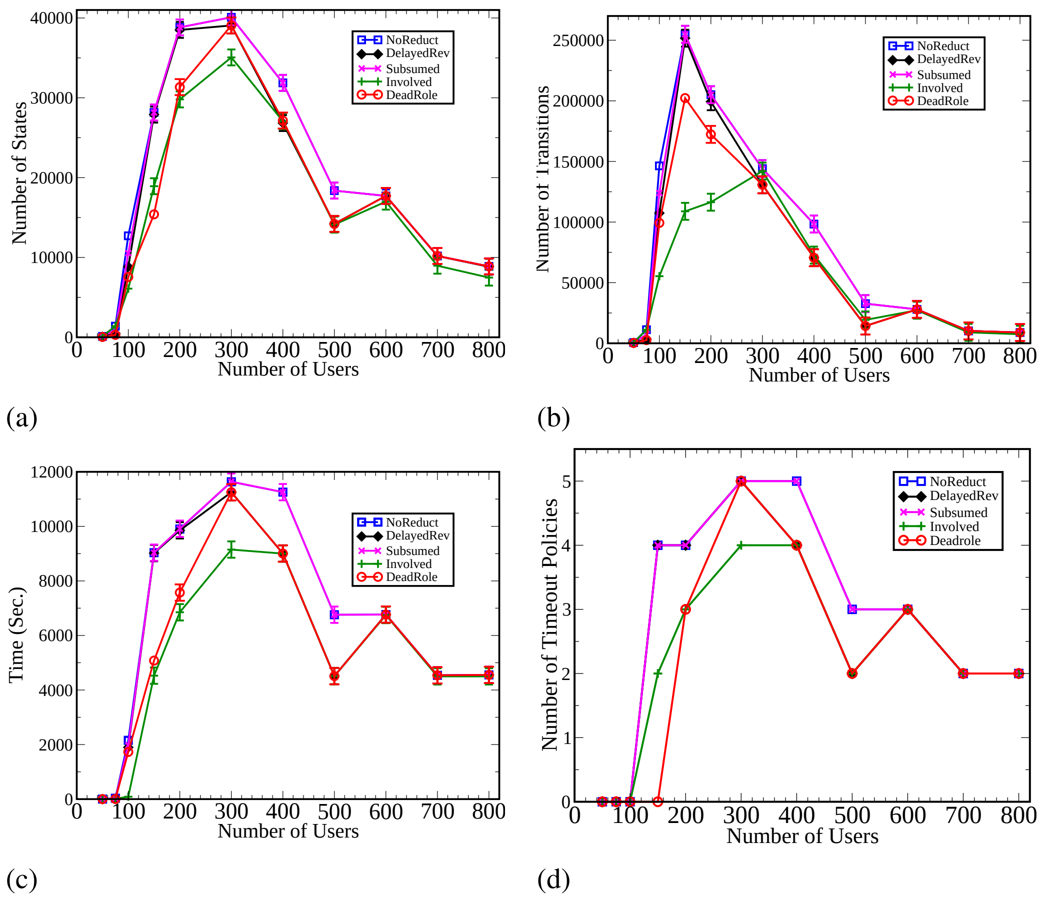


Figure 10: Experimental results of NoReduct and four least effective reductions; error bars indicate that the reported average is a lower bound on the actual average.

In our experiments, a run terminates as soon as the goal is reached (when the analysis result is true) or no more transitions can be computed (when the analysis result is false). We do not include the hierarchical rule reduction in the figures as this reduction itself is not effective in our experiments, because all administrative roles in the university ARBAC policy that have junior roles are mixed roles. In addition, the static involved-user reduction and the spare-user reduction cannot be applied together. AllReduct in Figure 11 uses the static involved-user reduction, which generates slightly fewer states and transitions than using the spare-user reduction for the reason given in Section 3.4.

The timeout threshold in our experiments is 5 hours. If a run on a particular query does not terminate in 5 hours, then we use the smallest number of states and transitions that had been explored when the timeout occurred in the computation of the average numbers of states and transitions explored for the current set of queries (corresponding to a given number of users and combination of reductions). Thus, in those cases, the reported averages are *lower bounds* on the actual averages that would have been obtained if we let the program always run to completion. In Figures 10 and 11, we use the *error bar* to indicate that the reported average is a lower bound on the actual average.

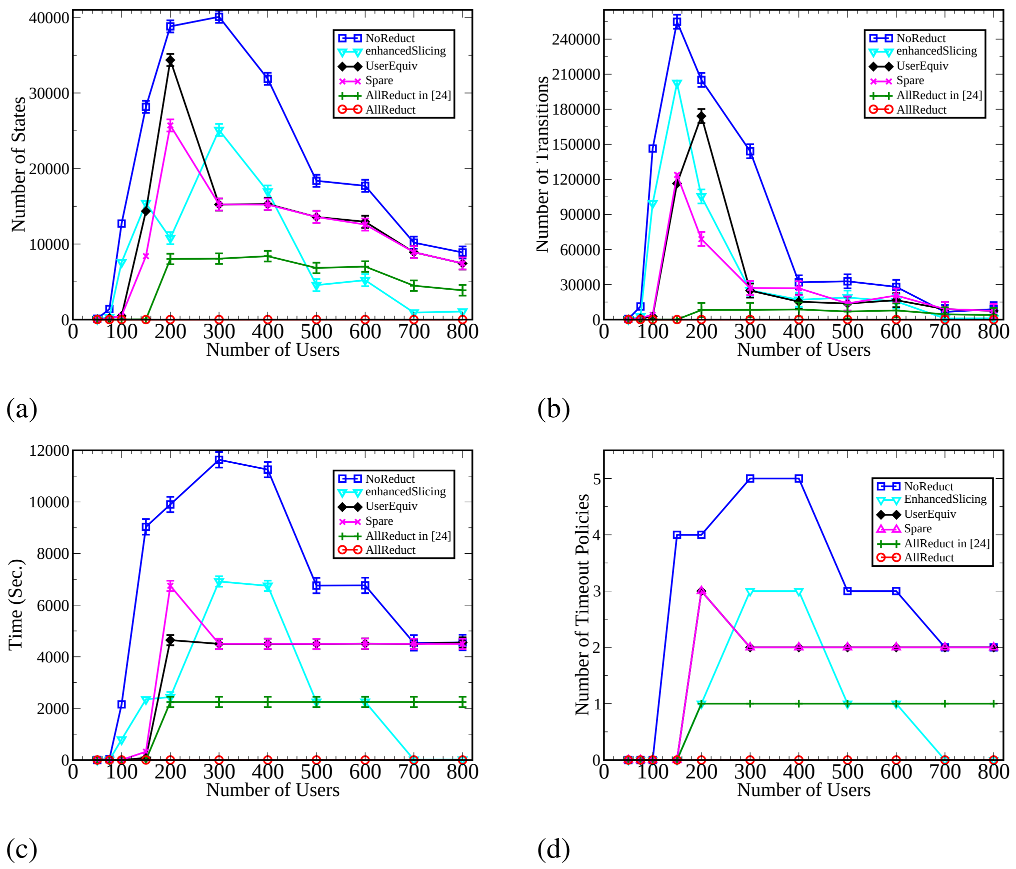


Figure 11: Experimental results of NoReduct and four most effective reductions; error bars indicate that the reported average is a lower bound on the actual average.

For example, we see from Figures 10(d) and 11(d) that, when the number of users is 200 or more, all variants except AllReduct timeout for 1 to 5 out of 8 policies.

While all reductions improve the performance, their effectiveness varies for different queries. The user equivalent set reduction performs the best for most policies and the subsumed-user reduction is the least effective (effective for only 75 and 100 users). Our experimental results also show that, although the static involved-user reduction does not reduce the number of states and transitions for 50 users, the execution time is less, because no transitions are performed on users who are eliminated. In addition, applying the spare-user reduction alone is more effective than applying the static involved-user reduction alone.

Note that the execution time of most variants for 400 users is smaller than that for 300 users. This is because the newly added users enable new transitions, which turns the analysis result for one of the 8 policies from “false” to ”true” and the program terminates as soon as the goal is reached. For the same reason, the execution time of most variants for 500 users is smaller than that for 400 users, and the execution time for 700 users is smaller than that for 600 users.

Integrating all reductions leads to a very effective solution. When the problem becomes difficult for the baseline algorithm to solve, AllReduct achieves an improvement of four orders of magnitude in execution time and terminates in all experiments. Our experimental results also show that AllReduct performs slightly better than Allreduct in [24] for 50–150 users and performs significantly better than Allreduct in [24] for 200–800 users. Further, integrating the two-stage slicing with the spare-user reduction reduces the number of administrative roles that need to be assigned to non-target users during analysis from 13 to 2-4.

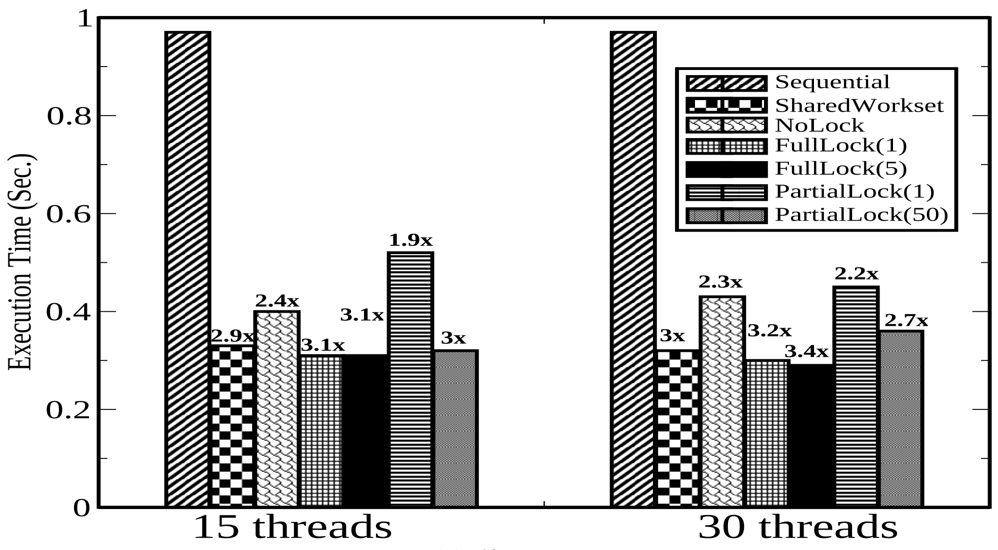
Performance results of parallel algorithms Figure 12 gives the execution time of our parallel analysis algorithms without reductions – SharedWorkset (Algorithm 5), NoLock, PartialLock(1), PartialLock(50), FullLock(1), and FullLock(5) – with 15 and 30 threads. FullLock(n) represents FullLock with the user-specified bound n. PartialLock(n) represents PartialLock in which a thread checks whether its neighbor is idle after computing every n transitions. The number atop each bar is the speedup of the corresponding parallel algorithm over the sequential algorithm. For example, Figure 12(a) shows that, with 50 users and 15 threads, SharedWorkset is 2.9 times faster than the sequential algorithm. Each The reported data were obtained on a computer with two 2.4GHz Quad-Core AMD Opteron Processors and 16GB RAM running Ubuntu 3.2.0. A 32-bit Ubuntu kernel was installed, so each process is limited to access only 4GB memory.

On average, FullLock performs the best, followed by PartialLock, NoLock, and SharedWorkset. The running times of FullLock(1) and FullLock(5) are almost the same. PartialLock(50) performs slightly better than PartialLock(1) for 50 and 75 users, but slightly worse for 100 users. FullLock and SharedWorkset with 30 threads outperform those with 15 threads, because the threads often wait for locks to access worksets in FullLock and SharedWorkset, and hence more CPU cores are utilized with 30 threads than 15 threads. The running times of NoLock and PartialLock with 15 threads are close to those with 30 threads.

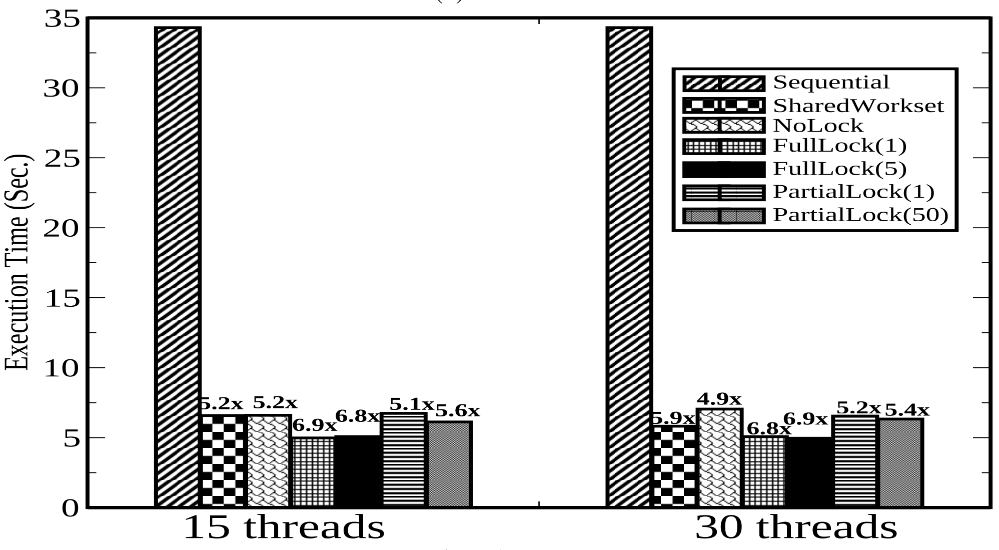
We have also measured the total number of times a thread accesses other threads' workset in FullLock. Our experimental results show that with 15 threads, the number of accesses to other threads' worksets is significantly less than that with 30 threads (by a factor of 64.65% to 83.59%). This is because, with fewer threads, there are more states in each workset and hence the chance that the workset becomes empty is smaller. Our experimental results also show that, with 30 threads, allowing a thread to take more than one state significantly reduces the average number of accesses to other threads' worksets (reduced by $4.97\% - 64.39\%$), but does not have a significant effect on the running time. This indicates that only a small fraction of the total execution time is spent in accessing other threads' worksets.

6 Related Work

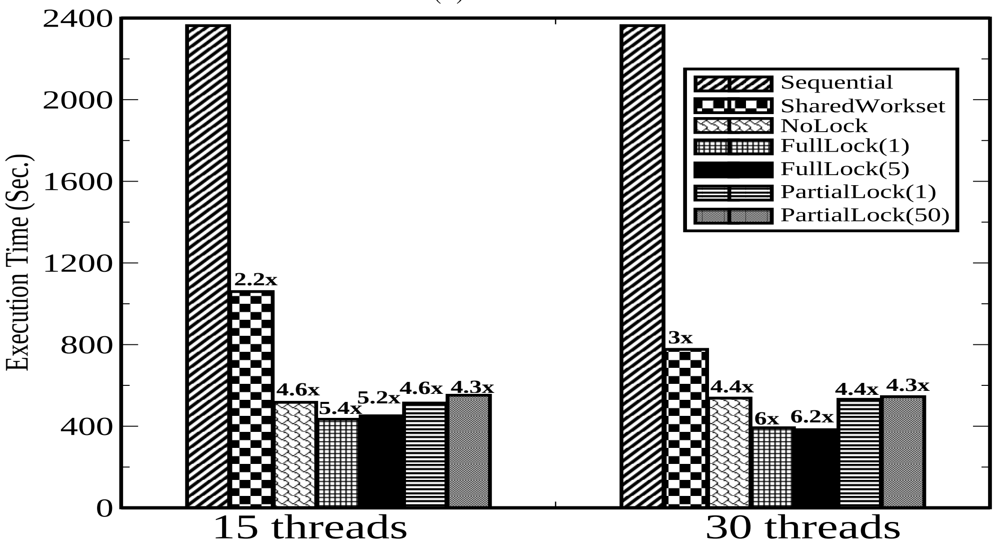
A number of researchers have studied user-role reachability analysis of ARBAC. Schaad et al. [19] applied the Alloy analyzer [12] to check the separation of duty properties for ARBAC97; they did not consider preconditions for any operations. Li et al.



(a) 50 users



(b) 75 users



(c) 100 users

Figure 12: Execution time of parallel algorithms.

[16] presented algorithms and complexity results for various analysis problems for two restricted versions of ARBAC97, called AATU and AAR; they did not consider negative preconditions. Jayaraman et al. [14] presented an abstraction refinement mechanism for detecting errors in ARBAC policies. Alberti et. al [1] developed a symbolic backward algorithm for analyzing Administrative Attribute-based RBAC policies, in which the policy and the query are encoded into a Bernays-Shonfinkel-Ramsey first order logic formulas. Becker [3] proposed a language *DYNPAL* for specifying dynamic authorization policies, which is more expressive than ARBAC, and presented techniques for analyzing *DYNPAL*. Sasturkar et al. [18] showed that user-role reachability analysis of ARBAC is PSPACE-complete, and presented algorithms and complexity results for ARBAC analysis subject to a variety of restrictions. Stoller et al. [20] presented algorithms for analyzing parameterized ARBAC. Gofman et al. [9] presented algorithms for analyzing evolving ARBAC. Uzun et al. [23] developed algorithms for analyzing temporal role-based access control models. However, none of the above works consider analysis of ARBAC without separate administration.

Several researchers have considered analysis of ARBAC without separate administration. Stoller et al. [22] provided fixed-parameter tractable algorithms for ARBAC with and without the separate administration restriction. Their algorithm for analyzing ARBAC without separate administration is exponential in the number of users in the policy, which is usually large in practice. Our work significantly improves the scalability of their algorithm by reducing the number of ARBAC rules and users considered during analysis. Ferrara et al [4] converted ARBAC policies to imperative programs and applied abstract-interpretation techniques to analyze the converted programs. However, if the goal is reachable, their approach cannot produce a trace which shows how the goal is reachable. Later, the same authors showed that if the goal is reachable in an ARBAC policy, then there exists a run of S with at most $|AR| + 1$ users in which the goal is reachable [5]. In this paper, we present three reductions that extend their work to further reduce the number of users considered during the analysis; our other reductions are different from their. In addition, none of the above works present parallel analysis algorithms.

A number of researchers considered analysis of fixed security policy [13, 15, 10, 11], analysis of a single change to a fixed policy, or analysis of differences between two fixed policies [15, 6]. However, none of these works consider analysis of ARBAC.

7 Conclusion and Future Work

This paper considers user-role reachability analysis without the separate administration restriction, which was shown to be PSPACE-complete in general. We present new analysis techniques with the goal of finding an efficient solution to the problem. Our techniques focus on reducing the number of ARBAC rules and users considered during analysis and delaying unnecessary computations. We also present parallel algorithms that speed up the analysis on multi-core systems. Experiments with a university ARBAC policy show that our techniques significantly reduce the analysis time. In the future, we plan to develop symbolic analysis algorithms to implicitly search the state space with a potential to further improve the performance of the analysis.

Acknowledgement: This work was supported in part by NSF Grants CNS-0855204 and CNS-0831298. We thank Kyoung-Don Kang for providing feedback on parallel algorithms and Dulcinea Chau for her contribution to the implementation of parallel algorithms.

References

[1] F. Alberti, A. Armando, and S. Ranise. Efficient symbolic automated analysis of administrative attribute-based RBAC-policies. In *ACM Symposium on Information, Computer and Communications Security*, pages 165–175, 2011.

[2] A. N. S. I. (ANSI). Role-based access control. ANSI INCITS Standard 359-2004, Feb. 2004.

[3] M. Y. Becker. Specification and analysis of dynamic authorisation policies. In *22nd IEEE Computer Security Foundations Symposium (CSF)*, pages 203 – 217, 2009.

[4] A. L. Ferrara, P. Madhusudan, and G. Parlato. Security analysis of role-based access control through program verification. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 113–125, 2012.

[5] A. L. Ferrara, P. Madhusudan, and G. Parlato. Policy analysis for self-administrated role-based access control. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 432 – 447, 2013.

[6] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *International Conference on Software Engineering (ICSE)*, pages 196–205, 2005.

[7] M. Gofman, R. Luo, J. He, Y. Zhang, and P. Yang. Incremental information flow analysis of role based access control. In *International Conference on Security and Management (SAM)*, pages 397–403, 2009.

[8] M. Gofman, R. Luo, A. Solomon, Y. Zhang, P. Yang, and S. Stoller. RBAC-PAT: A policy analysis tool for role based access control. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 46–49, 2009.

[9] M. Gofman, R. Luo, and P. Yang. User-role reachability analysis of evolving administrative role based access control. In *European Symposium on Research in Computer Security (ESORICS)*, pages 455 – 571, 2010.

[10] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in Security-Enhanced Linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[11] K. Irwin, T. Yu, and W. H. Winsborough. On the modeling and analysis of obligations. In *ACM Conference on Computer and Communications Security (CCS)*, pages 134–143, 2006.

[12] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *International Conference on Software Engineering (ICSE)*, pages 730–733, June 2000.

[13] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, 1997.

[14] K. Jayaraman, M. Tripunitara, V. Ganesh, M. Rinard, and S. Chapin. Mohawk: Abstraction-refinement and bound-estimation for verifying access control policies. *ACM Transaction on Information and System Security (TISSEC)*, 15(4):18:1–18:28, Apr. 2013.

[15] S. Jha and T. Reps. Model-checking SPKI-SDSI. *Journal of Computer Security*, 12:317–353, 2004.

[16] N. Li and M. V. Tripunitara. Security analysis in role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 9(4):391–420, Nov. 2006.

[17] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and Systems Security (TISSEC)*, 2(1):105–135, Feb. 1999.

[18] A. Sasturkar, P. Yang, S. D. Stoller, and C. Ramakrishnan. Policy analysis for administrative role based access control. *Theoretical Computer Science (TCS)*, 412(44):6208–6234, 2011.

[19] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 13–22, 2002.

[20] S. D. Stoller, P. Yang, M. I. Gofman, and C. Ramakrishnan. Symbolic reachability analysis for parameterized administrative role-based access control. *Journal of Computers & Security*, pages 148–164, 2011.

[21] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. An ARBAC policy for a university, `http://www.cs.sunysb.edu/~stoller/ccs2007/university-policy.txt`, 2007.

[22] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *14th ACM Conference on Computer and Communications Security (CCS)*, pages 445–455, 2007.

[23] E. Uzun, V. Atluri, S. Sural, J. Vaidya, G. Parlato, A. L. Ferrara, and M. Parthasarathy. Analyzing temporal role based access control models. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 177–186, 2012.

[24] P. Yang, M. Gofman, and Z. Yang. Policy analysis for administrative role based access control without separate administration. In *27th IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSEC)*, pages 49–64.