Policy Analysis for Administrative Role Based Access Control without Separate Administration

Ping Yang, Mikhail Gofman, Zijiang Yang

Policy Analysis for Administrative Role Based Access Control without Separate Administration

Ping Yang[1] and Mikhail Gofman[2] and Zijiang Yang[3]

[1] Dept. of Computer Science, State University of New York at Binghamton, NY, USA
[2] Dept. of Computer Science, California State University at Fullerton, CA, USA
[3] School of Computer Science and Engineering, Xi'an University of Technology, China
Dept. of Computer Science, Western Michigan University, MI, USA

Abstract. Access control is widely used in large systems for restricting resource access to authorized users. In particular, role based access control (RBAC) is a generalized approach to access control and is well recognized for its many 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 a number of 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

Access control is widely used for restricting resource access to authorized users. In particular, role based access control (RBAC) [2] is broadly recognized as a generalized approach to access control that has many 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. $UA \subseteq U \times R$ represents the user-role assignment relation and $PA \subseteq P \times R$ represents 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 (ARBAC'97) [17] defines administrative roles and specifies how members of each administrative role can change the RBAC policy. ARBAC specifies *user-role administration* which controls the way changes are made to the user-role assignments. This control is enforced by two types of rules: (1) $can\_assign(r_a, c, r_t)$ that grants an administrative role r_a permission to assign a target role r_t to any user who satisfies the precondition c, and (2) $can\_revoke(r_a, r_t)$

that 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. ARBAC'97 requires *separate administration* [22], *i.e.*, administrative roles cannot be target roles in *can_assign* and *can_revoke* rules or appear in the preconditions. In the rest of this paper, we represent the precondition c as $P \wedge \neg N$ where P contains all positive preconditions and N contains all negative preconditions in c.

The correctness of ARBAC policies is critical to system security because any design flaws and human specification errors in ARBAC may result in the leak of confidential data to unauthorized users. Large organizations may have large ARBAC policies. In such organizations, 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 the desirable correctness properties.

This paper considers the user-role reachability analysis of ARBAC [22], which asks "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 roles in the goal"? Since many 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 is crucial for ARBAC analysis.

Researchers have shown that 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 that assumes separate administration. By disallowing an administrative role to serve as the target role in any of the ARBAC rules, 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 $AdmissionComittee$, which can in turn assign any user to role $Student$. Formally, this specification translates to the rules $can\_assign(DeptChair, Faculty, AdmissionComittee)$ and $can\_assign(AdmissionCommittee, true, Student)$, which do not satisfy the separate administration restriction.

Analysis of ARBAC without separate administration is significantly more challenging because we need to 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 number of users and mixed roles. That is, the algorithm is exponential to the number of users and mixed roles, but is polynomial to the size of the policy when the number of users and mixed roles is fixed. However, since the number of users is usually large in large organizations, the algorithm does not scale well when analyzing ARBAC policies in such organizations. For example, we have applied this algorithm to analyze a university ARBAC policy containing 150 users and the program failed to terminate within 12 hours for 3 out of the 10 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 – optimized slicing (Section 3.1) and hierarchical rule reduction (Section 3.4) – to reduce the number of ARBAC rules considered during analysis.
- We develop a user equivalent set reduction technique that reduces the number of users considered during analysis (Section 3.2).
- We propose a lazy reduction technique that delays performing unnecessary transitions (Section 3.3).
- We present several parallel algorithms, which speed up the analysis on multi-core or multi-processor platforms (Section 4).
- We evaluate the effectiveness of our reduction techniques and our parallel algorithms on an ARBAC policy representing a 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 our 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 analysis of ARBAC [22] asks: "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"? Let UA_0 be a set of all user-role assignments in ϕ. The user-role reachability analysis instance is represented as a tuple $I = \langle UA_0, u_t, \psi, goal \rangle$.

Stoller et al. [22] presented an algorithm for analyzing ARBAC without separate administration, which is formalized in Algorithm 1. The algorithm is fixed parameter tractable with respect to the number of users and mixed roles. A role is *negative* if it appears negatively in some precondition in the policy; other roles are *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; other roles are *non-positive*. A role that is both negative and positive is a *mixed* role. Note that their algorithm is applied to ARBAC without role hierarchy; ARBAC with role hierarchy can be converted to the corresponding non-hierarchical policy using the algorithm in [18]. Let $I = \langle UA_0, u_t, \psi, goal \rangle$ be a user-role reachability analysis problem instance. The algorithm works as follows.

First, the algorithm performs a slicing transformation (function $slicing$ in Line 3), which back-chains along the ARBAC rules to identify roles and rules relevant to the goal, and then eliminates the irrelevant ones. Function $slicing$ takes into account whether a role appears positively or negatively in the policy, and computes a set Rel_+ of positive roles and a set Rel_- of negative roles that are relevant to the goal. A set $RelRule$ of relevant rules is computed as a collection of all can_assign rules whose

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

$Processed = Rel_+ = Rel_- = \emptyset$; $RelRule = \emptyset$;
procedure $analysis(UA_0, u_t, \psi, goal)$
$(Rel_+, Rel_-, RelRule) = slicing(UA_0, \psi, goal)$; $W = Reached = \{closure(UA_0)\}$;
if $goal \subseteq \{r \mid (u_t, r) \in closure(UA_0)\}$ **then** return true; **end if**
while $W \neq \emptyset$ **do**
  remove a state s from W;
  for all $can\_assign(r_a, P \wedge \neg N, r) \in RelRule$ **do**
   for all (user $u \in U$) **do**
    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')
    then $s' = \text{closure}(s \cup \{(u, r)\})$; add transition $s \overset{ua(r_a,u,r)}{\rightarrow} s'$ to G;
      if $goal \subseteq \{r | (u_t, r) \in s'\}$ **then** return true; **end if**
      if $s' \notin Reached$ **then** $W = W \cup \{s'\}$; $Reached = Reached \cup \{s'\}$ **end if**
    end if **end for** **end for**
  for all $(can\_revoke(r_a, r) \in RelRule)$
   for all (user $u \in U$)
    if $((u, r) \in s$ and $(u', r_a) \in s$ for some user $u')$
    then $s' = \text{closure}(s \setminus \{(u, r)\})$; add transition $s \overset{ur(r_a,u,r)}{\rightarrow} s'$ to G;
      if $goal \subseteq \{r \mid (u_t, r) \in s'\}$ **then** return true; **end if**
      if $s' \notin Reached$ **then** $W = W \cup \{s'\}$; $Reached = Reached \cup \{s'\}$ **end if**
    end if **end for** **end for**
end while
return false;
procedure $slicing(UA_0, \psi, goal)$
if $goal = \emptyset$ **then** return $(\emptyset, \emptyset, \emptyset)$ **end if**
$Processed = Processed \cup goal$; $R_+ = goal$; $R_- = \emptyset$; $Rule = \emptyset$;
for all $can\_assign(r_a, P \wedge \neg N, r) \in \psi$ where $r \in goal$ **do**
  $(R_1, R_2, R_3) = slicing(UA_0, \psi, (\{r_a\} \cup P) \setminus Processed)$; $R_+ = R_+ \cup R_1$;
  $R_- = R_- \cup N \cup R_2$; $Rule = Rule \cup \{can\_assign(r_a, P \wedge N, r)\} \cup R_3$;
end for
$RelRev = \{can\_revoke(r_a, r) \in \psi \mid r \in R_-\}$; $Rule = Rule \cup RelRev$;
for all $can\_revoke(r_a, r) \in RelRev$ where $r_a \notin Processed$ **do**
  $(R_4, R_5, R_6) = slicing(UA_0, \psi, \{r_a\})$; $R_+ = R_+ \cup R_4$;
  $R_- = R_- \cup R_5$; $Rule = Rule \cup R_6$
end for
return $(R_+, R_-, Rule)$
procedure $closure(s)$
$s_1 = s$;
for all $can\_assign(r_a, P \wedge \neg N, r) \in RelRule$ **do**
 for all user $u \in U$ **do**
  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')
  then $s_1 = s_1 \cup (u, r)$; **end if** **end for** **end for**
if $s == s_1$ **then** return s_1; **else** return $closure(s_1)$;

targets are in Rel_+ and all can_revoke rules whose targets are in Rel_-; only rules in $RelRule$ need to be applied during analysis.

Next, the algorithm constructs a *reduced transition 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*. The 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).

In [22], they have also identified a condition called the *hierarchical role assignment (HRA)*, 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 analysis problem for this policy with the initial RBAC policy $UA_0 = \{(u_1, r_1), (u_1, r_3), (u_2, r_2), (u_2, r_8), (u_3, r_2), (u_3, r_8), (u_t, r_6)\}$, the target user u_t, 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 both an administrative role in rule 2 and a target role in rule 4.

First, the algorithm performs slicing to compute a set Rel_+ of positive relevant roles and a set Rel_- of negative relevant roles as follows. Initially, Rel_+ contains all roles in the goal, i.e. r_5. Since the target role of rule 2 is r_5, the algorithm adds positive preconditions and administrative role of rule 2, i.e. r_4, r_3, and r_6, to Rel_+. The algorithm then processes rules 1 and 3, whose target roles are r_4 and r_3, respectively, adds their positive preconditions and administrative roles, i.e. r_2, r_6, and r_1, to Rel_+, and adds their negative preconditions, i.e. r_3, to Rel_-. Repeat this process until all roles in Rel_+ are processed, which results in $Rel_+ = \{r_1, r_2, r_3, r_4, r_5, r_6, r_8\}$ and $Rel_- = \{r_3\}$. The set of mixed roles is $Rel_+ \cap Rel_- = \{r_3\}$; other roles are both positive and non-negative. $RelRule$ contains rules 1, 2, 3, 4, and 7.

Next, the algorithm computes the initial state $closure(UA_0)$. Since rule 3 is enabled from UA_0 and r_4 is a non-negative role, (u_t, r_4) is added to UA_0 through an invisible transition. The algorithm then computes all states reachable from $closure(UA_0)$ using

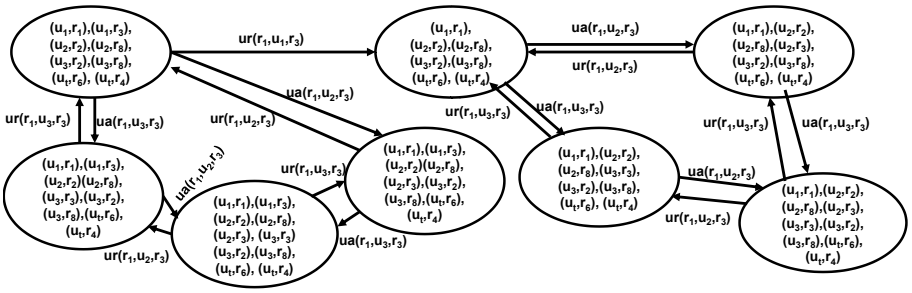

Fig. 1. Graph constructed in Example 1 using the algorithm in [22].

rules in $RelRule$. The resulting graph is given in Figure 1. Because the graph does not contain (u_t, r_5), the goal is not reachable. □

3 Reduction Techniques

The analysis algorithm described in Section 2, although simple, does not scale well for policies containing a large number of users. Let $I = \langle UA_0, u_t, \psi, goal \rangle$ be a user-role reachability analysis problem instance. In this section, we present a number of techniques for reducing the number of users and ARBAC rules considered during analysis.

3.1 Optimized Slicing

In this section, we present an approach to reduce the number of roles processed during slicing, and hence reduce the number of relevant rules computed.

We say that a role is irrevocable if there does not exist a can_revoke rule that revokes the role. For the target user u_t, we apply function $slicing$ defined in Algorithm 1 to perform slicing, except that Line 27 in the algorithm is replaced with the following:

$S = \{r \mid r \in (P \cup \{r_a\}) \wedge (r \text{ is nonnegative or irrevocable}) \wedge (u_t, r) \in UA_0\}$;
$(R_1, R_2, R_3) = slicing(UA_0, \psi, (\{r_a\} \cup P) \setminus S) \setminus Processed)$;

Similarly, Line 32 of Algorithm 1 is replaced with the following:

$S = \{r_a \mid (r_a \text{ is nonnegative or irrevocable}) \wedge (u_t, r_a) \in UA_0\}$;
$(R_4, R_5, R_6) = slicing(UA_0, \psi, (\{r_a\} \setminus S))$; $R_+ = R_+ \cup R_4$;

Basically, prior to slicing, we collect a set of nonnegative and irrevocable roles in the ARBAC policy. During slicing, we do not slice nonnegative or irrevocable roles assigned to the target user in the initial policy UA_0. This is safe because such roles will not be revoked during the analysis and hence we do not need to reassign such roles to

Algorithm 2 An Optimized Slicing Algorithm for Non-Target Users.

1: $Processed = \emptyset$;
2: **procedure** $optslicing(UA_0, \psi, goal)$
3: **if** $(goal == \emptyset)$ **then** return $(\emptyset, \emptyset, \emptyset)$; **end if**
4: $Processed = Processed \cup goal$; $R_+ = goal$; $R_- = \emptyset$; $Rule = \emptyset$;
5: **for all** $can\_assign(r_a, P \wedge \neg N, r)$ where $(u_t, r) \in goal$ **do**
6: **if** $((u, r_a) \in UA_0$ for some user u and (r_a is non-negative or irrevocable)) **then**
7: $R_1 = R_2 = R_3 = \emptyset$;
8: **else** $(R_1, R_2, R_3) = slicing(UA_0, \psi, \{r_a\} \setminus Processed)$; **end if**
9: $S = \{r \mid r \in P \wedge (r$ is non-negative or irrevocable$) \wedge (u_t, r) \in UA_0\}$;
10: $(R'_1, R'_2, R'_3) = optslicing(UA_0, \psi, (P \setminus S) \setminus Processed)$;
11: $R_+ = R_+ \cup S \cup R_1 \cup R'_1$; $R_- = R_- \cup N \cup R_2 \cup R'_2$; $Rule = Rule \cup R_3 \cup R'_3$;
12: **end for**
13: $RelRev = \{can\_revoke(r_a, r) \mid r \in R_-\}$; $Rule = Rule \cup RelRev$;
14: **for all** $can\_revoke(r_a, r) \in RelRev$ **do**
15: **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**
16: $(R_4, R_5, R_6) = slicing(\psi, \{r_a\})$;
17: $R_+ = R_+ \cup R_4$; $R_- = R_- \cup R_5$; $Rule = Rule \cup R_6$
18: **end if**
19: **end for**
20: return $(R_+, R_-, Rule)$;

the target user. 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 negative roles remains unchanged.

For non-target users, it is sufficient to apply only rules that assign such users to administrative roles, which have permission to assign the target user u_t to the goal. The pseudocode is given in Algorithm 2.

The reduction is given in Lines 6–10 and 15–16 of Algorithm 2. For every $can\_assign(r_a, P \wedge \neg N, r)$ where $r \in goal$, we check if r_a is a nonnegative or irrevocable role assigned to a user in UA_0. If so, we do not slice r_a; otherwise, we apply function $slicing$ defined in Algorithm 1 to slice r_a (Lines 6–8). This is different from Algorithm 1, in which r_a is always sliced. Next, we compute a set S of all nonnegative or irrevocable roles in P that are assigned to the target user in UA_0, and for every rule whose target role is in $P \setminus S$, we recursively call function $optslicing$ to slice the administrative roles of such rules (Lines 9–10). Note that we do not slice roles in P for non-target users, while Algorithm 1 does. Finally, for every $can\_revoke(r_a, r)$, if r_a is a nonnegative or irrevocable role assigned to some user in UA_0, we do not slice r_a (Lines 15–16).

Example: Consider the ARBAC policy and the query in Example 1. First, we compute a set of relevant roles and rules for the target user u_t using our optimized slicing mechanism. Since r_6 is a non-negative role assigned to u_t in UA_0, we do not slice r_6. Therefore, for the target user, $Rel_+ = \{r_1, r_2, r_3, r_4, r_5, r_6\}$, $Rel_- = \{r_3\}$, and $RelRule = \{1, 2, 3, 7\}$. Next, we compute a set of relevant roles and rules for non-

target users using our optimized slicing mechanism. Only administrative roles that have permissions to assign the target user to the goal, i.e., r_6 and r_1, need to be sliced. Since r_6 and r_1 are non-negative roles assigned to u_t and u_1 in UA_0, respectively, we do not slice these two roles. As a result, for non-target users, $Rel_+ = \{r_1, r_6\}$, $Rel_- = \{r_3\}$, and $RelRule = \emptyset$. This means that there is no need to assign roles to non-target users. The transition graph constructed with the optimized slicing 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 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 non-target users assigned distinct sets of roles. Our technique is based on a notion of user equivalent set defined below.

Definition 1 *The user equivalent set w.r.t 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 $u \in Uset_i$, $Rset_i = \{r|(u, r) \in s\}$.*

The user equivalent set w.r.t a state s is basically an alternative representation of s, in which all users assigned the same set of roles are grouped together. Let G_{ue} be the transition graph constructed using the user equivalent set representation. There is a transition $ue(s) \xrightarrow{\alpha} ue(s')$ in G_{ue} if and only if there is a transition $s \xrightarrow{\alpha} s'$ 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 represent the transition graph constructed with the user equivalent set reduction.

The correctness of the reduction is formalized in Theorem 1. 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 $s_1\delta = s_2$. For example, $\{(\{u_1, u_t\}, \{r_1, r_2\}), (\{u_2\}, \{r_2\})\} \equiv \{(\{u_2, u_t\}, \{r_1, r_2\}), (\{u_1\}, \{r_2\})\}$ holds because there exists a substitution $\delta = \{u_1/u_2, u_2/u_1\}$ such that $\{(\{u_1, u_t\}, \{r_1, r_2\}), (\{u_2\}, \{r_2\})\}\delta = \{(\{u_2, u_t\}, \{r_1, r_2\}), (\{u_1\}, \{r_2\})\}$.

Theorem 1 *Let $I = \langle UA_0, u_t, \psi, goal\rangle$ be a user-role reachability analysis instance, and G_{redue} and G_{ue} be transition graphs constructed for I with and without using the user equivalent set reduction. The goal is reachable in G_{ue} iff the goal is reachable in G_{redue}.*

Example: Consider the user-role reachability analysis instance in Example 1. Since non-target users u_2 and u_3 are assigned the same set of roles in the initial state, we

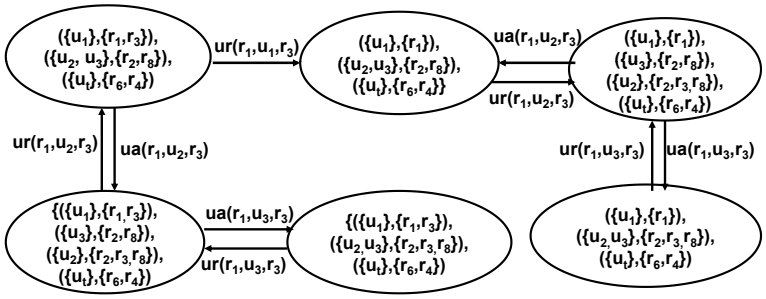

Fig. 2. The transition graph constructed with the user equivalent set reduction.

need to perform only transitions for u_2 or u_3, but not both, from the initial state. This is different from Algorithm 1, which 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 2, which contains 6 states and 9 transitions, i.e., 25% reduction on states and 47% on transitions.

Optimization: We can reduce the size of the state 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 ($u_t \in Uset$) or 0 ($u_t \notin Uset$).

3.3 Delayed Revocation

In this section, we propose to reduce the size of the transition graph by delaying transitions that can neither enable new transitions in s nor be disabled by any transitions. Formally, a transition $s \xrightarrow{ur(r_a,u,r)} s'$ is not performed from s (i.e. is delayed) if

1. $trans(s) = trans(s') \cup \{ur(r_a, u, r)\}$ where $trans(s)$ and $trans(s')$ are sets of all visible transitions enabled from s and s', respectively,
2. $s \setminus s' = \{(u, r)\}$,
3. r_a is non-negative or irrevocable.

Rules 1 and 2 specify that $s \xrightarrow{ur(r_a,u,r)} s'$ does not enable new visible and invisible transitions, respectively. Rule 3 specifies that $s \xrightarrow{ur(r_a,u,r)} s'$ cannot be disabled by other transitions.

Given a state s, we compute transitions that can be delayed in s as follows. First, we perform all ua transitions that assign users in s to roles. Next, for every ur transition that is enabled in s, we check if the transition enables any transition. If so, we perform the transition from s. Otherwise, we add the transition to a set $Delayed$. Since performing ur transitions may enable new ua and ur transitions, after all can_revoke rules are processed, we compute new transitions and check if any transitions in $Delayed$ enable other transitions. If so, such transitions are performed from s and are removed from $Delayed$. Repeat the above process until no new transitions are computed.

The correctness of the delayed revocation reduction is formalized in Theorem 2.

Theorem 2 *Let $I = \langle UA_0, u_t, \psi, goal \rangle$ be a user-role reachability analysis instance, $s_0 = closure(UA_0)$, and G_{dr} and G be transition graphs constructed for I with and without the delayed revocation reduction. The goal is reachable in G iff the goal is reachable in G_{dr}.*

Example: Consider the user-role reachability analysis instance in Example 1. Since transition $ur(r_1, u_1, r_3)$ 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. The transition graph constructed contains 4 states and 8 transitions, i.e., 50% reduction on the number of states and 47% reduction on the number of transitions.

3.4 Hierarchical Rule Reduction

Hierarchical rule reduction avoids considering rules whose administrative preconditions are junior to non-negative or irrevocable administrative roles in UA_0. This is safe because senior roles inherit all administrative permissions of their junior roles, and non-negative/irrevocable roles are never revoked during analysis. This 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 problem instance in Example 1 and the role hierarchy $r_1 \succeq r_2$. The following three rules are added after the policy is transformed into the non-hierarchical one: $can\_assign(r_1, \{r_8, r_1\} \wedge \neg\emptyset, r6)$, $can\_assign(r_1, \{r_6\} \wedge \neg\emptyset, r_7)$, 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 useful 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.

4 Parallel Analysis Algorithm

Multi-core processors are becoming pervasive. In order for software applications to benefit from the continued exponential throughput advances in new computer systems, it is important to parallelize the applications. In this section, we extend Algorithm 1 to perform analysis in parallel. The pseudocode of our parallel algorithm is given in Algorithm 3.

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). Finally, each thread t_i removes one state from 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 state $Reached$ if the target states are not already in $Reached$ (Lines 11–20). Since multiple threads may access $Reached$ at the same time, $Reached$ needs to be protected by locks in order to ensure the correct execution of the program. Obviously, locking and unlocking $Reached$ every time a thread accesses $Reached$ imposes high overhead. To reduce the time spent on waiting for locks to access $Reached$, we implemented $Reached$

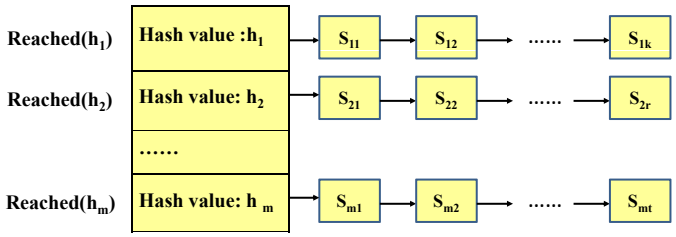

Fig. 3. Implementation of the set of reachable states $Reached$.

Algorithm 3 User-Role Reachability Analysis Algorithm in [22].

1: $Reached = W = Rel_+ = Rel_- = \emptyset$; $RelRule = \emptyset$; $done = 0$;
2: **procedure** $mcanalysis(UA_0, u_t, \psi, goal)$
3: $(Rel_+, Rel_-, RelRule) = slicing(UA_0, \psi, goal)$; $init = closure(UA_0)$;
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 $s \overset{ua(r_a,u,r)}{\rightarrow} s'$
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;

as a hashtable shown in Figure 3. The hashtable is partitioned into multiple regions $Reached(h_1), \ldots, Reached(h_m)$; $Reached(h_i)$ stores a set of states whose hash values are h_i. Once a thread computes a transition $s \overset{\alpha}{\rightarrow} s'$, 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 $Reached(h(s'))$. The above approach enables two threads to access two different regions in $Reached$ simultaneously. Our experimental results show that locking $Reached(h(s))$ instead of $Reached$ significantly improves the performance. This is because threads access $Reached$ very frequently and checking

if 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.

It is also possible to reduce the time spent on waiting for locks to access the workset W by having each thread to have its own workset. Below, we present three approaches to minimizing (or completely removing) the number of operations performed on locking/unlocking W.

- **NoLock:** In this approach, each thread is not allowed to access 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 the requirement for locking, but may result in idle threads (due to empty workset).
- **FullLock:** In this approach, a thread is allowed to access other threads' workset to retrieve a state to process, if the thread's workset is empty. This approach ensures that all threads will be approximately equally busy, but it requires to lock the workset every time the workset is accessed.
- **PartialLock:** In this approach, whenever a thread t_i computes a new transition, it checks if thread $t_{(i-1) \ mod \ n}$ is idle. If so, it locks the workset of $t_{(i-1) \ mod \ n}$, adds the target state to the workset, unlocks the workset, and starts $t_{(i-1) \ mod \ n}$. The advantage of this approach is that locking is only needed when t_i adds a state to $t_{(i-1) \ mod \ n}$'s workset. This approach has limitation that each thread t_i has to frequently check if $t_{(i-1)modn}$ is sleeping.

Discussion: Two threads can safely access the same region in $Reached$ simultaneously if neither thread adds a state to or removes a state from the same region. Thus, in some cases, it may be possible to improve the performance by replacing mutual exclusion locks on $Reached$ with *reader-writer locks*. Unlike a mutual exclusion lock, which prevents all concurrent accesses to a critical region, a reader-writer lock allows multiple threads performing read operations to enter critical region. Our experiments, however, show that such optimization does not yield performance improvement (in fact, it often causes performance degradation). This is because multiple threads rarely access the same region in $Reached$ simultaneously during analysis, and reader-writer locks, due to their complexity, impose greater overheads than mutual exclusion locks.

5 Performance Results

This section evaluates the effectiveness of our reduction techniques and our parallel algorithms using the university ARBAC policy developed in [22] and the university RBAC policy developed in [7]. All reported data were obtained on a 2.4GHz 2 Quad-Core AMD Opteron Processor with 16GB RAM running Ubuntu 3.2.0.

The university RBAC and ARBAC policies contain 845 users, 32 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, honor program director, etc. A sample can_assign rule is: *the honors program director can assign an undergraduate*

50 non-target users					
	NoReduct	OptSlice	DelayedRev	UserEquivSet	AllReduct
State	111	45	54	15	4
Transition	620	264	278	61	9
Time	0.97	0.41	0.57	0.13	0.09
75 non-target users					
	NoReduct	OptSlice	DelayedRev	UserEquivSet	AllReduct
State	24909	245	6393	273	5
Transition	214165	2168	42718	3222	10
Time	34.30	6.18	10.99	0.15	0.09
100 non-target users					
	NoReduct	OptSlice	DelayedRev	UserEquivSet	AllReduct
State	12706	7552	7855	39	6
Transition	145115	99520	107323	225	11
Time	2363	2029.26	2166.81	0.81	0.1

Table 1. Performance of analysis algorithms without reduction, with a single reduction, and with all reductions.

student to the honors student role. A sample user-role reachability problem instance 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 honor student role?*

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 honor 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 if users in $\{u_1, \ldots, u_n, u_t\}$ together can assign u_t to both honor student role and role r.

Effectiveness of Reduction Techniques Table 1 gives the the size of the transition graph and the execution time for three sets of experiments with different numbers of randomly chosen non-target users (50, 75 or 100). Each data point reported in the table is an average over 8 randomly generated queries. The five columns represent reduction techniques applied during the experiments: with no reduction (NoReduct), with optimized slicing (OptSlice), with user equivalent set (UserEquivSet), with delayed revocation (DelayedRev), and with all reductions (AllReduct). Note that we do not include the hierarchical rule reduction in the table as it is not effective in our experiments. This is because all administrative roles in the university policy that have junior roles are mixed roles and remain mixed after applying all reductions.

We observe that, while all reduction techniques improve the performance, their effectiveness varies under different queries. UserEquivSet performs the best for all three sets of experiments and DelayedRev is the least effective. 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. In addition, when the number of non-target users is 150, NoReduct fails to complete 3 of the 8 queries within 12 hours, whereas the average analysis time of AllReduct is only 0.1 seconds.

50 non-target users									
		15 threads				30 threads			
	NoReduct	SharedWorkset	NoLock	FullLock	PartialLock	SharedWorkset	NoLock	FullLock	PartialLock
Time	0.97	0.33	0.40	0.32	0.52	0.32	0.43	0.34	0.45
75 non-target users									
		15 threads				30 threads			
	NoReduct	SharedWorkset	NoLock	FullLock	PartialLock	SharedWorkset	NoLock	FullLock	PartialLock
Time	34.30	6.58	6.60	5.85	6.74	5.82	7.04	5.80	6.54
100 non-target users									
		15 threads				30 threads			
	NoReduct	SharedWorkset	NoLock	FullLock	PartialLock	SharedWorkset	NoLock	FullLock	PartialLock
Time	2363	1059.73	517.09	436.87	513.46	776.34	537.68	407.53	531.83

Table 2. Performance of the parallel algorithm without reduction.

Performance Results of Parallel Algorithms Table 2 gives the execution time of four parallel analysis algorithms without reductions – SharedWorkset (Algorithm 3), NoLock, PartialLock, and FullLock – with 15 and 30 threads. The results show that, on average, FullLock performs the best, followed by PartialLock, NoLock, and SharedWorkset. FullLock and SharedWorkset with 30 threads outperform those with 15 threads, because the threads often wait for locks to access the worksets in FullLock and SharedWorkset, and hence more CPU cores are utilized with 30 threads than 15 threads. NoLock and PartialLock with 15 threads outperform those with 30 threads, because the threads do not or only occasionally wait for locks in NoLock and PartialLock, and hence the CPU cores are mostly utilized with 15 threads.

6 Related Work

A number of researchers have studied user-role reachability analysis of ARBAC. Schaad et al. [20] 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. [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. [19] 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. [21] 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 administrative restriction. Their algorithm for analyzing

ARBAC without separate administration is exponential to the number of users in the policy, which is usually large in practice. Our work significantly improved 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 $|\text{administrative roles}| + 1$ users in which the goal is reachable [5]. However, their algorithm and reduction techniques are different from ours. Their techniques can be combined with ours to further reduce the analysis time. In addition, none of the above works present parallel analysis algorithms.

A number of researchers have 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 them consider analysis of ARBAC.

7 Conclusion and Future Work

This paper considers the 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 a practical solution to the problem. Our techniques focus on reducing the number of ARBAC rules and users considered during analysis and delaying unnecessary computations. We have also presented a number of parallel algorithms that speed up the analysis on multi-core systems. The experimental results on 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 user-role reachability analysis.

Acknowledgement: This work was supported in part by NSF Grant CNS-0855204. We thank Kyoung-Don Kang for providing feedbacks 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)*, 2009.
4. A. L. Ferrara, P. Madhusudan, and G. Parlato. Security analysis of role-based access control through program verification. In *Computer Security Foundations Symposium*, pages 113–125, 2012.
5. A. L. Ferrara, P. Madhusudan, and G. Parlato. Policy analysis for self-administrated role-based access control. In *to appear, International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 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*, 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*, 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*, 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*, pages 134–143, 2006.
12. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. pages 730–733, June 2000.
13. S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Symposium on Security and Privacy*, pages 31–42, 1997.
14. K. Jayaraman, V. Ganesh, M. Tripunitara, M. Rinard, and S. Chapin. Automatic error finding for access control policies. In *Proceedings of 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.
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*, 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. In *IEEE Computer Security Foundations Workshop*, 2006.
19. A. Sasturkar, P. Yang, S. D. Stoller, and C. Ramakrishnan. Policy analysis for administrative role based access control. *Theoretical Computer Science*, 412(44):6208–6234, 2011.
20. 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*, pages 13–22, 2002.
21. 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.
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*, pages 177–186, 2012.