Efficient Policy Analysis for Evolving Administrative Role Based Access Control [1]

Mikhail I. Gofman[1] and Ping Yang[2]

[1] Dept. of Computer Science, California State University of Fullerton, CA, 92834

[2] Dept. of Computer Science, State University of New York at Binghamton, NY, 13902

Abstract Role Based Access Control (RBAC) has been widely used for restricting resource access to only authorized users. Administrative Role Based Access Control (ARBAC) specifies permissions for administrators to change RBAC policies. Due to complex interactions between changes made by different administrators, it is often difficult to comprehend the full effect of ARBAC policies by manual inspection alone. Policy analysis helps administrators detect potential flaws in the policy specification.

Prior work on ARBAC policy analysis considers only static ARBAC policies. In practice, ARBAC policies tend to change over time in order to fix design flaws or to cope with the changing requirements of an organization. Changes to ARBAC policies may invalidate security properties that were previously satisfied. In this paper, we present incremental analysis algorithms for evolving ARBAC. Our incremental algorithms determine if a change may affect the analysis result, and if so, use the information of the previous analysis to incrementally update the analysis result. To the best of our knowledge, these are the first known incremental algorithms in literature for ARBAC analysis. Detailed evaluations show that our incremental algorithms outperform the non-incremental algorithm in terms of execution time at a reasonable cost of increased disk space consumption.

Key words: security, administrative role-based access control, security policy analysis

1 Introduction

Role Based Access Control (RBAC) [20] has been widely used for restricting resource access to only authorized users. In large organizations, RBAC policies are often managed by multiple administrators with varying levels of authority. Administrative Role Based Access Control'97 (ARBAC97) [23] defines administrative roles and controls changes to RBAC policies.

Correct understanding of ARBAC policies is critical for maintaining the security of underlying systems. However, ARBAC policies in large organizations are usually large and complex. Consequently, it is difficult to comprehend the full effect of such policies by manual inspection alone. Automated policy analysis helps administrators understand ARBAC policies and detect potential flaws in the policies. A number of analysis techniques have been developed for user-role reachability analysis of ARBAC [24, 26, 15], which asks, given an RBAC policy and an ARBAC policy, a set of

[1] This Paper is an expanded version of a paper entitled "User-Role Reachability Analysis of Evolving Administrative Role Based Access Control" presented at the 15th European Symposium on Research in Computer Security (ESORICS), 2010.

‡ Corresponding author: Ping Yang, email: pyang@binghamton.edu

administrators A, a target user u_t, and a set of roles (called the "goal"), is it possible for administrators in A to assign the target user u_t to roles in the goal? It has been shown that the user-role reachability analysis for ARBAC (with fixed role hierarchy) is PSPACE-complete [24, 15]. Some other analysis problems, such as permission-role reachability, user-role availability, and role-role containment, can be solved by reducing them to the user-role reachability analysis problem [26].

Prior work on ARBAC analysis considers static ARBAC policies. In practice, ARBAC policies tend to change over time in order to correct design flaws or to cope with changing requirements of an organization. Changes to an ARBAC policy may invalidate security properties that were previously satisfied. Therefore, the policy needs to be re-analyzed to verify its overall correctness.

Contributions: In this paper, we present algorithms for analyzing evolving ARBAC. Our algorithms use the information of the previous analysis to incrementally update the analysis result. In a predominant majority of cases, the information obtained from the previous analysis can be used to update the result more quickly than a complete re-analysis. In a small minority of cases, a complete re-analysis cannot be avoided. To the best of our knowledge, these are the first known algorithms in literature for analysis of evolving ARBAC policies. Our contributions are as follows:

- We propose three incremental forward analysis algorithms – IncFwd1, IncFwd2, and LazyInc – for user-role reachability analysis of ARBAC with separate administration restriction (i.e., administrative roles and regular roles in the policy are disjoint). These algorithms are based on the forward algorithm in [26], which constructs a *reduced transition graph* from an ARBAC policy; the analysis result is true if and only if the goal is a subset of some state in the graph. IncFwd1 determines if a change to the policy may affect the analysis result, and if so, performs non-incremental analysis; otherwise, the algorithm simply returns the previous analysis result. IncFwd2 incrementally updates the transition graph constructed in the previous analysis: if a change to the policy may affect the transition graph, IncFwd2 updates the graph by pruning states and transitions that are invalidated by the change and adding new states and transitions that are enabled by the change. Our experimental data shows that updating the transition graph is faster than reconstructing it from scratch. LazyInc reuses the transition graph constructed in the previous analysis, but delays the update to the graph until an operation, which may affect the analysis result, is performed.
- We have developed a backward algorithm for incremental user-role reachability analysis of ARBAC with separate administration. This algorithm is based on the backward algorithm in [26], which consists of two stages. In the first stage, a goal-directed backward search is performed, which constructs a directed graph G_b. In the second stage, a forward search is performed on G_b to determine if the goal is reachable. Our algorithm reuses G_b as well as the result computed in the second stage. If a *can_revoke* rule (which specifies the authority to remove a user from a role) is added or deleted, G_b remains the same and the algorithm simply updates the second stage. If a *can_assign* rule (which specifies the authority to add a user to a role) is added or deleted, the algorithm determines if the change may affect G_b, and if so, updates G_b as well as the second stage.

- We have developed incremental algorithms for permission-role reachability, user-role availability, role-role containment, weakest precondition, and dead role analysis. Permission-role reachability, user-role availability, and role-role containment problems present a straightforward reduction to user-role reachability, while dead role analysis and weakest precondition problems require more complex approaches.
- We have proposed an incremental forward algorithm for analyzing ARBAC without separate administration restriction. Examples of such policies include the university and health-care policies developed in [26]. Without this restriction, the analysis must consider administrative actions on all users, not just the target user.
- We have implemented the incremental user-role reachability analysis algorithms presented in this paper. The experimental data shows that our incremental analysis algorithms outperform the non-incremental algorithms in terms of the execution time at a reasonable cost of increased disk space consumption.

Organization: The rest of the paper is organized as follows. Section 2 provides a brief overview of RBAC, ARBAC, and the user-role reachability analysis algorithms in [26]. Sections 3 and 4 present incremental analysis algorithms for ARBAC with and without separate administration, respectively. Incremental algorithms for permission-role reachability, user-role availability, role-role containment, weakest precondition, and dead role analysis are given in Section 5. The related work is discussed in Section 6 and our concluding remarks appear in Section 7.

2 Preliminaries

2.1 Role Based Access Control

Access control policies specify who can access which resources. In Role Based Access Control (RBAC) [20], roles are assigned to users and permissions are associated with roles. Formally, an RBAC policy is defined as 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 a set of user-role assignments. $(u, r) \in UA$ specifies that user u is a member of role r. For example, $(John, Faculty) \in UA$ specifies that user John is a member of role $Faculty$.
- $PA \subseteq P \times R$ is a set of permission-role assignments. $(p, r) \in PA$ specifies that role r has granted permission p. For example, $(Registercourse, Student) \in PA$ specifies that every member of role $Student$ has permission to register for courses.

RBAC supports role hierarchy, which is a partial order on the set of roles. For example, role hierarchy relation $Deptchair \succeq Faculty$ specifies that role $Deptchair$ is senior to role $Faculty$, i.e. every member of role $Deptchair$ is implicitly a member of role $Faculty$, and every member of role $Deptchair$ has all permissions of role $Faculty$.

2.2 Administrative Role Based Access Control

Administrative Role-Based Access Control'97 (ARBAC97) [23] defines administrative roles and controls changes to RBAC policies. An ARBAC97 policy consists of three main parts: (1) the user-role administration (URA) which controls changes to the user-role assignments, (2) the permission-role administration (PRA) which controls changes to the permission-role assignments, and (3) the role-role administration (RRA) which controls changes to the role hierarchy. In this paper, we consider analysis of URA and PRA policies, but not RRA, i.e., we do not consider changes to the role hierarchy. The URA and PRA policies are defined below.

URA policy: The URA policy controls changes to the set of user-role assignments. Authority to assign a user to a role is specified using the *can_assign* relation. Each *can_assign* relation is represented as $\mathit{can\_assign}(r_a, c, r_t)$ where r_a is an administrative role, r_t is the target role, and c is the prerequisite condition (or precondition) of the rule. 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. For example, $\mathit{can\_assign}(\mathit{Faculty}, \mathit{Grad} \wedge \neg\mathit{TA}, \mathit{RA})$ specifies that the administrative role *Faculty* has the permission to assign a user who is a member of role *Grad* but not a member of role *TA*, to role *RA*. In 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.

Given an RBAC policy ϕ and a *can_assign* rule $\mathit{can\_assign}(r_a, P \wedge \neg N, r_t)$, a user u satisfies the precondition $P \wedge \neg N$ iff for every $r \in P$, u is a member of role r in ϕ, and for every $r' \in N$, u is not a member of role r' in ϕ. A $ua(r_a, u, r)$ action specifies that an administrator in role r_a assigns user u to role r. This action is enabled in an RBAC policy $\phi = \langle U, P, R, \mathit{UA}, \mathit{PA}\rangle$ iff there exist $\mathit{can\_assign}(r_a, P \wedge \neg N, r)$ and a user $u' \in U$ such that u' is a member of r_a in ϕ, u is not a member of r in ϕ, and u satisfies $P \wedge \neg N$. Execution of this action transforms the RBAC policy ϕ to $\langle U, P, R, \mathit{UA} \cup \{(u, r)\}, \mathit{PA}\rangle$.

Authority to revoke a user from a role is specified using the *can_revoke* relation. Each *can_revoke* relation is represented as $\mathit{can\_revoke}(r_a, r_t)$, which specifies that the administrative role r_a has the permission to remove users from the target role r_t. A $ur(r_a, u, r)$ action specifies that an administrator in role r_a revokes user u from role r. This action is enabled in an RBAC policy $\phi = \langle U, P, R, \mathit{UA}, \mathit{PA}\rangle$ iff there exist $\mathit{can\_revoke}(r_a, r)$ and a user $u' \in U$ such that u' is a member of r_a in ϕ and u is a member of r in ϕ. Execution of this action transforms the RBAC policy ϕ to $\langle U, P, R, \mathit{UA} \setminus \{(u, r)\}, \mathit{PA}\rangle$.

PRA policy: The PRA policy controls changes to the set of permission-role assignments *PA*. Authority to assign a permission to a role is specified using the *can_assign_p* relation, which is defined in the same way as the *can_assign* relation, except that users are replaced with permissions. Similarly, authority to revoke a permission from a role is specified using the *can_revoke_p* relation, which is defined in the same way as the *can_revoke* relation, except that users are replaced with permissions.

ARBAC97 requires that administrative roles and regular roles are disjoint, i.e., administrative roles do not appear in the preconditions and target roles of *can_assign* rules, as well as target roles of *can_revoke* rules. This restriction is called the *separate administration restriction* in [26]. In practice, not all ARBAC policies satisfy this restriction. For example, the university ARBAC policy and the healthcare ARBAC policy developed in [26] do not satisfy this restriction. In this paper, we consider analysis of ARBAC with and without this restriction.

2.3 User-Role Reachability Analysis of ARBAC With Separate Administration

The user-role reachability analysis problem asks, given an initial RBAC policy ϕ, an ARBAC policy ψ, a set of administrators A, a target user u_t, and a set of roles (called the "goal"), is it possible for administrators in A to assign the user u_t to roles in the goal, under the restrictions imposed by ψ? We use a tuple $\langle UA_0, \psi, goal\rangle$ to represent the user-role reachability analysis problem instance, where UA_0 is a set of all user-role assignments in the initial RBAC policy ϕ.

Sasturkar et al. [24] showed that, under the separate administration restriction, the problem can be simplified as follows: (1) Because each user's role memberships are controlled independently of other users' role memberships, it is sufficient to consider only role assignments of the target user u_t in ϕ. (2) Because administrative roles and regular roles are disjoint, it suffices to consider only ARBAC rules whose administrative roles are in A or are junior to roles in A. As a result, administrative roles in A can be merged into a single administrative role and all other administrative roles can be eliminated. With the above simplification, the *can_assign* rule is simplified as $can\_assign(c, r_t)$, *can_revoke* rule is simplified as $can\_revoke(r_t)$, $ua(r_a, u, r_t)$ is simplified as $ua(r_t)$, $ur(r_a, u, r_t)$ is simplified as $ur(r_t)$, and the user-role reachability analysis problem is simplified as $\langle R_{0t}, \psi, goal\rangle$ where $R_{0t} = \{r \mid (u_t, r) \in UA_0\}$. In addition, they have also shown that, when the role hierarchy is fixed (i.e., when the RRA policy is not considered), user-role reachability analysis problem for hierarchical ARBAC can be transformed into that for non-hierarchical ARBAC. Therefore, in this paper, we present algorithms for only non-hierarchical ARBAC policies.

Our incremental algorithms are developed upon two analysis algorithms in [26]. Below, we summarize these two algorithms.

The forward algorithm In the forward algorithm, roles are classified into *negative* and *non-negative* roles, and *positive* and *non-positive* 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 or appears positively in some precondition in the policy; other roles are *non-positive.* A role that is both negative and positive is a *mixed* role.

Given a user-role reachability analysis problem instance $I = \langle R_{0t}, \psi, goal\rangle$, the algorithm first performs a slicing transformation, which computes a set of roles that are relevant to the goal. Let $Rel_+(I)$ and $Rel_-(I)$ denote the set of positive and negative roles relevant to the goal, respectively. $Rel_+(I)$ and $Rel_-(I)$ are formally computed as follows.

- $r \in Rel_+(I)$ if $r \in goal$.
- $r \in Rel_+(I)$ if there exists $can\_assign(P \wedge \neg N, r_1) \in \psi$ such that $r_1 \in Rel_+(I)$ and $r \in P$.
- $r \in Rel_-(I)$ if there exists $can\_assign(P \wedge \neg N, r_1) \in \psi$ such that $r_1 \in Rel_+(I)$ and $r \in N$.

A role r is a relevant mixed role if $r \in Rel_+(I) \cap Rel_-(I)$. A role r is an irrelevant role if $r \notin (Rel_+(I) \cup Rel_-(I))$. The algorithm then computes a set of relevant rules *RelRule*, which consists of *can_assign* rules whose target roles are in $Rel_+(I)$ and *can_revoke* rules whose target roles are in $Rel_-(I)$.

Next, the algorithm constructs a *reduced transition graph* $G(I)$ from I using rules in *RelRule* and performs analysis based on $G(I)$. Each state in $G(I)$ is a set of roles assigned to the target user and each transition describes an allowed change to the state defined by the ARBAC policy ψ. A transition either has the form $ua(r)$ which assigns the target user to role r or has the form $ur(r)$ which revokes the target user from role r. The following reductions are applied: (1) Irrelevant roles and revocable non-positive roles are removed from R_{0t}; the resulting set of roles is represented as $InitRm_I(R_{0t})$; (2) Transitions that revoke non-negative roles or add non-positive roles are prohibited because they do not enable any other transitions; (3) Transitions that add non-negative roles or revoke non-positive roles are *invisible*; other transitions are *visible*. Invisible transitions will not disable any other transitions. Invisible transitions together with a visible transition form a single composite transition.

Let closure(s, I) be the largest state that is reachable from state s by performing invisible transitions enabled from s using rules in ψ. The algorithm constructs $G(I)$ as follows. First, the algorithm computes closure$(InitRm_I(R_{0t}), I)$, which is the initial state of $G(I)$. The algorithm then computes states reachable from closure$(InitRm_I(R_{0t}), I)$: given a state s, there is a transition $s \overset{ua(r)}{\rightarrow}$ closure$(s \cup \{r\}, I)$ if there exists $can\_assign(c, r) \in \psi$ such that r is a mixed role, s does not contain r, and s satisfies c. There is a transition $s \overset{ur(r)}{\rightarrow}$ closure$(s \setminus \{r\}, I)$ if there exists $can\_revoke(r) \in \psi$ such that r is a mixed role and s contains r. The algorithm returns true iff a state containing all roles in the goal is computed.

Example 1. *Consider the following ARBAC policy ψ and the reachability analysis problem instance $I = (\{r_1\}, \psi, \{r_5\})$.*

1. $can\_assign(true, r_1)$	*5. $can\_assign(r_4 \wedge \neg r_3, r_5)$*
2. $can\_assign(r_1, r_2)$	*6. $can\_assign(r_2, r_6)$*
3. $can\_assign(r_2, r_3)$	*7. $can\_assign(\neg r_1, r_7)$*
4. $can\_assign(r_3, r_4)$	*no roles except r_1 can be revoked*

First, the algorithm performs slicing to compute a set $Rel_+(I)$ of positive relevant roles and a set $Rel_-(I)$ of negative relevant role: $Rel_+(I) = \{r_1, r_2, r_3, r_4, r_5\}$ and $Rel_-(I) = \{r_3\}$. The set of relevant mixed roles is $Rel_+(I) \cap Rel_-(I) = \{r_3\}$. r_1, r_2, r_4 and r_5 are roles that are both positive and non-negative. Since r_6 and r_7 are not in $Rel_+(I)$, rules 6 and 7 are not relevant to the goal and hence are eliminated. Since $r_1 \notin Rel_-(I)$, the rule for revoking r_1 is not relevant to the goal and hence is eliminated.

Next, the algorithm constructs a reduced transition graph $G(I)$ by computing $closure(\{r_1\}, I) = \{r_1, r_2\}$ and a set of states reachable from $closure(\{r_1\}, I)$. The graph $G(I)$ is given in Figure 1. The mixed role r_3 is added to states through visible transitions. Roles r_2 and r_4 are added to states through invisible transitions. Because role r_5 does not appear in the graph, the goal is not reachable. □

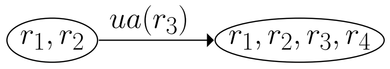


Fig. 1. The reduced transition graph constructed in Example 1 using the forward algorithm.

The backward algorithm The backward algorithm in [26] comprises two stages. The first stage performs a backward search from the goal and constructs a directed graph G_b. Each node in G_b is a set of roles. There is an edge $V_1 \overset{\langle P\wedge\neg N,T\rangle}{\rightarrow} V_2$ in G_b if there exists a rule $\mathit{can\_assign}(P \wedge \neg N, T) \in \psi$ such that starting from V_1, revoking all roles that appear in N and adding T, results in node V_2.

The second stage performs a forward search from the initial nodes in G_b to determine if the goal is reachable. A node V is an initial node in G_b if $V \subseteq R_{0t}$. The goal is reachable if there exists a *feasible plan* in G_b. A plan corresponds to a path of edges in G_b from the initial node to the node containing the goal, which is constructed as follows: starting from the initial state, for each edge $V_1 \overset{\langle P\wedge\neg N,T\rangle}{\rightarrow} V_2$ in G_b, a plan contains a transition $s_1 \overset{\alpha}{\rightarrow} s_2$ where s_1 is a state corresponding to V_1, and α is a sequence of actions that revokes roles in $s_1 \cap N$ and adds T. A plan is feasible if it does not revoke irrevocable roles.

To correctly check if G_b contains a feasible plan, each node in G_b is associated with a set of *additional irrevocable roles* (*airs*). $\mathit{airs}(V)$ represents a set of irrevocable roles that appear in the state corresponding to node V, but not in V itself. Formally, given an edge $V_1 \overset{\langle P\wedge\neg N,T\rangle}{\rightarrow} V_2$, $\mathit{airs}(V_2)$ is defined as:

$$\{S \cup ((\mathit{UA}_1 \setminus \mathit{UA}_2) \cap \mathit{Irrev}) \mid S \in \mathit{airs}(\mathit{UA}_1), ((\mathit{UA}_1 \cap \mathit{Irrev}) \cup S) \cap N = \emptyset\}$$

Irrev is a set of all irrevocable roles in the policy. $((\mathit{UA}_1 \cap \mathit{Irrev}) \cup S) \cap N = \emptyset$ is the condition that needs to be satisfied in order to add T to the state. The goal is reachable iff $\mathit{airs}(\mathit{goal}) \neq \emptyset$.

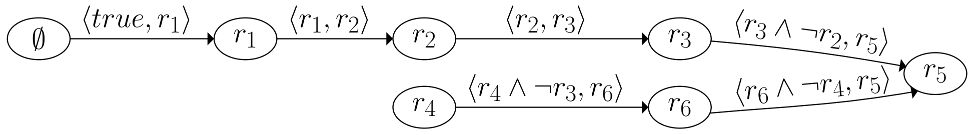
Example 2. *Consider the following ARBAC policy ψ' and the reachability analysis problem instance $I' = (\{r_4\}, \psi', \{r_5\})$.*

1. $\mathit{can\_assign}(\mathit{true}, r_1)$	*5. $\mathit{can\_assign}(r_4 \wedge \neg r_3, r_6)$*
2. $\mathit{can\_assign}(r_1, r_2)$	*6. $\mathit{can\_assign}(r_6 \wedge \neg r_4, r_5)$*
3. $\mathit{can\_assign}(r_2, r_3)$	*7. $\mathit{can\_assign}(r_2, r_7)$*
4. $\mathit{can\_assign}(r_3 \wedge \neg r_2, r_5)$	*All roles are revocable*

The graph computed in the first stage of the backward algorithm is given in Figure 2. There are two initial nodes: $\{r_4\}$ and $\emptyset$. Assume that the algorithm starts from $\{r_4\}$ and computes the airs of nodes reachable from $\{r_4\}$. Since r_4 is revocable, $\mathit{airs}(\{r_4\}) = \{\emptyset\}$. The algorithm then computes $\mathit{airs}(\{r_6\})$ and $\mathit{airs}(\{r_5\})$, which are $\{\emptyset\}$. Because $\mathit{airs}(\{r_5\})$ is not empty, the goal is reachable. Since nodes $\emptyset$, $\{r_1\}$, $\{r_2\}$, and $\{r_3\}$ are not processed, $\mathit{airs}(\emptyset) = \mathit{airs}(\{r_1\}) = \mathit{airs}(\{r_2\}) = \mathit{airs}(\{r_3\}) = \emptyset$. □

2.4 User-Role Reachability Analysis without Separate Administration

Without separate administration restriction, we need to consider administrative actions on any user, not only the target user. This is because, without this restriction, a user (target or non-target user) can be assigned to an administrative role, which



$\emptyset$
$\langle true, r_1\rangle$
r_1
$\langle r_1, r_2\rangle$
r_2
$\langle r_2, r_3\rangle$
r_3
$\langle r_3 \wedge \neg r_2, r_5\rangle$
r_5
r_4
$\langle r_4 \wedge \neg r_3, r_6\rangle$
r_6
$\langle r_6 \wedge \neg r_4, r_5\rangle$

Fig. 2. The graph constructed in Example 2 using the backward algorithm.

may enable this user to change the role assignments of the target user. For example, consider the following RBAC and ARBAC policies:

$(u, DeptChair) \in UA, (u, Faculty) \in UA$
$can\_assign(DeptChair, Faculty, GradDirector)$
$can\_assign(GradDirector, true, GradStudent)$

The above policy does not satisfy the separate administration restriction since role *GradDirector* is both a regular role in the first *can_assign* rule and an administrative role in the second *can_assign* rule. The above policy would enable u to assign himself/herself to role *GradDirector* and then assign any user to role *GradStudent*.

Stoller et. al [26] presented a forward algorithm for ARBAC without separate administration. This algorithm is similar to the forward algorithm in Section 2.3 except that (1) We need to process administrative roles during slicing; (2) Each state in the reduced transition graph is a set of user-role relations of the form (u, r); (3) given a state s, there is a transition $s \overset{ua(r_a,u,r)}{\rightarrow}$ closure$(s \cup \{(u, r)\}, I)$ if there exists $can\_assign(r_a, c, r) \in \psi$ such that $(u_a, r_a) \in s$ for some user u_a, r is a mixed role, s does not contain (u, r), and user u satisfies c in state s. There is a transition $s \overset{ur(r_a,u,r)}{\rightarrow}$ closure$(s \setminus \{(u, r)\}, I)$ if there exists $can\_revoke(r_a, r) \in \psi$ such that $(u_a, r_a) \in s$ for some user u_a, r is a mixed role, and s contains (u, r).

Example 3. *Consider the following ARBAC policy ψ'' and query $I = \langle\{(u_1, r_1), (u_2, r_2), (u_3, r_4)\}, \psi'', \{(u_3, r_5)\}\rangle$*

1. $can\_assign(r_1, r_2, r_3)$
2. $can\_assign(r_3, r_4, r_5)$
3. $can\_assign(r_1, r_6 \wedge \neg r_3, r_4)$

The policy does not satisfy the separate administration restriction because role r_3 is both an administrative role (i.e. appears as the administrative precondition of rule 2) and a regular role (i.e. appears as a postcondition of rule 1).

First, the algorithm performs slicing transformation to compute a set $Rel_+(I)$ of positive relevant roles and a set $Rel_-(I)$ of negative relevant roles: $Rel_+(I) = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ and $Rel_-(I) = \{r_3\}$. The set of mixed roles is $Rel_+(I) \cap Rel_-(I) = \{r_3\}$ and all other roles are both positive and non-negative. All rules are relevant. Next, the algorithm computes $S = closure(\{(u_1, r_1), (u_2, r_2), (u_3, r_4)\}, I)$ and all states reachable from S. The resulting graph is given in Figure 3. (u_2, r_3) is added to the graph through a visible transition. (u_3, r_5) is added to the graph through an invisible transition. Because the graph contains (u_3, r_5), the goal is reachable. □

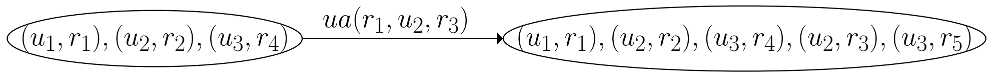

$(u_1, r_1), (u_2, r_2), (u_3, r_4)$
$ua(r_1, u_2, r_3)$
$(u_1, r_1), (u_2, r_2), (u_3, r_4), (u_2, r_3), (u_3, r_5)$

Fig. 3. The transition graph constructed in Example 3 using the forward algorithm for ARBAC without separate administration.

3 Incremental User-Role Reachability Analysis of ARBAC with Separate Administration

This section presents our incremental algorithms for user-role reachability analysis of ARBAC with the separate administration restriction.

3.1 Incremental Forward Algorithms

We present three forward algorithms – IncFwd1, IncFwd2, and LazyInc – for incremental reachability analysis of ARBAC. We consider the following changes: (1) adding a *can_assign* rule, (2) deleting a *can_assign* rule, (3) adding a *can_revoke* rule, and (4) deleting a *can_revoke* rule. Because ARBAC with role hierarchy can be transformed to ARBAC without role hierarchy, this paper considers ARBAC without role hierarchy.

IncFwd1 determines if a change may affect the analysis result, and if so, performs non-incremental analysis; otherwise, IncFwd1 returns the previous analysis result. IncFwd2 reuses the transition graph constructed in the previous analysis and incrementally updates the graph. LazyInc also reuses the graph constructed in the previous analysis, but it does not update the graph until an operation that may affect the analysis result is performed. IncFwd1 does not require additional disk space. IncFwd2 and LazyInc require additional disk space to store the transition graph computed, but are faster than IncFwd1. All three algorithms have the same worst-case complexity as the non-incremental algorithm.

Let $I = \langle R_{0t}, \psi, goal\rangle$ be a user-role reachability analysis problem instance and $G(I)$ be the transition graph constructed from I using the non-incremental forward algorithm in [26]. Below, we describe the three incremental analysis algorithms in detail.

3.2 Incremental Algorithm: IncFwd1

IncFwd1 is developed based on the following two observations:

- If the analysis result of I is true, then the following changes will not affect the analysis result: (1) adding a *can_assign* rule; (2) adding a *can_revoke* rule; (3) deleting a *can_assign* rule whose target role is not in $Rel_+(I)$; and (4) deleting a *can_revoke* rule whose target role is not in $Rel_-(I)$. (1) and (2) do not affect the analysis result because, adding a *can_assign* or a *can_revoke* rule cannot disable any ARBAC rules applied in the previous analysis and hence the goal can be reached by applying rules in I. (3) and (4) do not affect the analysis result because *can_assign* rules whose target roles are not in $Rel_+(I)$ and *can_revoke* rules whose target roles are not in $Rel_-(I)$ are irrelevant to the goal, and hence the goal can be reached without applying such rules.
- If the analysis result of I is false, then the following changes will not affect the result: (1) deleting a *can_assign* rule; (2) deleting a *can_revoke* rule; (3) adding a

can_assign rule whose target role is not in $Rel_+(I)$; and (4) adding a *can_revoke* rule whose target role is not in $Rel_-(I)$. (1) and (2) do not affect the analysis result because, deleting a *can_assign* or a *can_revoke* rule cannot enable any ARBAC rules applied in the previous analysis and hence the goal still cannot be reached. (3) and (4) do not affect the analysis result because *can_assign* rules whose target roles are not in $Rel_+(I)$ and *can_revoke* rules whose target roles are not in $Rel_-(I)$ are irrelevant to the goal, and hence adding such rules does not help reach the goal.

IncFwd1 uses the slicing transformation result of the previous analysis to determine if a change may affect the analysis result. If so, IncFwd1 performs re-analysis using the non-incremental algorithm; otherwise, IncFwd1 incrementally updates the slicing result and returns the previous analysis result.

3.3 Incremental Algorithm: IncFwd2

IncFwd2 reuses the slicing transformation result and the transition graph computed in the previous analysis, and incrementally updates the graph. States whose outgoing transitions are not computed or not computed completely are marked as “UnProcessed”. If a goal that is reachable in I becomes unreachable after a change is made to the policy, IncFwd2 performs non-incremental analysis from states that were previously marked as “UnProcessed”. Let $G_{inc}(I)$ denote the transition graph computed by IncFwd2 for the problem instance I. Below, we describe IncFwd2 in detail.

Adding a can_revoke rule Suppose that $can\_revoke(T)$ is added to the policy ψ. Let $I_1 = \langle R_{0t}, \psi \cup \{can\_revoke(T)\}, goal\rangle$. Figure 4 gives the pseudocode for constructing graph $G_{inc}(I_1)$ from $G(I)$.

Since only *can_assign* rules are used during slicing, adding a *can_revoke* rule does not change the result of the slicing transformation. Thus, $Rel_+(I_1) = Rel_+(I)$ and $Rel_-(I_1) = Rel_-(I)$. If T is a mixed role relevant to the goal, the algorithm starts from the initial state *init* of $G(I)$ and, for every state containing T, adds a transition $ur(T)$ and marks new target state as ”UnProcessed” (lines 3 – 18). Otherwise, if T is in R_{0t} and is both negative and non-positive, the algorithm replaces *init* with $closure(init \setminus \{T\}, I_1)$ and propagates roles in $closure(init \setminus \{T\}, I_1) \setminus init$ to states reachable from *init* (lines 19–35). Because different states in $G_{inc}(I_1)$ may be computed from the same state in $G(I)$, the workset W_1 contains pairs of the form $\langle s, s'\rangle$ where $s \in G(I)$, $s' \in G_{inc}(I_1)$, and s' is computed from s. The algorithm then calls function *addNewTrans* to add new transitions enabled by the rule; this function returns true if a state containing the goal is reached (line 32). The above process is then repeated on states reachable from *init*. In other cases, the transition graph as well as the analysis result remain the same (line 36).

If the goal is reachable, the algorithm calls function *markUnproc* to mark states in $G_{inc}(I_1)$, whose outgoing transitions are not computed (i.e., the remaining states in W or W_1) or not computed completely (i.e., the state from which the goal is reached by a transition), as “UnProcessed”. Otherwise, the algorithm processes all states marked as ”UnProcessed” using the non-incremental algorithm (line 38).

Example 4. *Consider the analysis problem instance in Example 1.*

```
$init' = closure(InitRm_{I_1}(R_{0t}), I_1)$
if $goal \subseteq init'$ then $add\ init'\ to\ G_{inc}(I_1)$;  return $true$ endif
if $T \in Rel_+(I_1) \cap Rel_-(I_1)$
  $W = reached = \{init\}$
  while $W \neq \emptyset$ do
    $Remove\ s\ from\ W$
    for $s \overset{\alpha}{\to} s_1 \in G(I)$ do
      $add\ s \overset{\alpha}{\to} s_1\ to\ G_{inc}(I_1)$
      if $goal \subseteq s_1$ then $markUnproc(W \cup \{s\})$; return $true$ endif
      if $s_1 \notin reached$ then $W = W \cup \{s_1\}$;  $reached = reached \cup \{s_1\}$ endif
    endfor
    if $T \in s$ then
      $s' = closure(s \setminus \{T\}, I_1)$
      if $s' \notin G(I)$ then $markUnproc(\{s'\})$ endif
      $add\ s \overset{ur(T)}{\to} s'\ to\ G_{inc}(I_1)$
      if $goal \subseteq s'$ then $markUnproc(W)$; return $true$ endif
    endif
  endwhile
elseif $T \in (R_{0t} \cap Rel_-(I_1))$ and $T \notin Rel_+(I_1)$ then
  $W_1 = reached_1 = \{\langle init, init' \rangle\}$
  while $W_1 \neq \emptyset$ do
    $Remove\ \langle s, s' \rangle\ from\ W_1$
    for $(s \overset{\alpha}{\to} s_1) \in G(I)$ do
      $s'_1 = closure((s_1 \setminus \{T\}) \cup (s' \setminus s), I_1)$
      $add\ s' \overset{\alpha}{\to} s'_1\ to\ G_{inc}(I_1)$
      if $goal \subseteq s'_1$ then $markUnproc(\{s'_i | \langle s_i, s'_i \rangle \in W_1\} \cup \{s'\})$;  return $true$ endif
      if $\langle s_1, s'_1 \rangle \notin reached_1$ then
        $W_1 = W_1 \cup \{\langle s_1, s'_1 \rangle\}$;
        $reached_1 = reached_1 \cup \{\langle s_1, s'_1 \rangle\}$
      endif
    endfor
    if $addNewTrans(s, s') == true$ then
      $markUnproc(\{s'_i | \langle s_i, s'_i \rangle \in W_1\} \cup \{s'\})$; return $true$
    endif
  endwhile
else $return\ the\ analysis\ result\ of\ I$ endif
if $G_{inc}(I_1) == \emptyset$ then $add\ init'\ to\ G_{inc}(I_1)$;  return $false$ endif
$process\ all\ states\ marked\ UnProcessed\ with\ non-incremental\ alg.$
```

Fig. 4. Pseudocode for adding $can\_revoke(T)$

$$\begin{tikzcd}
r_1, r_2 \arrow[r, "ua(r_3)"] & r_1, r_2, r_3, r_4 \arrow[r, "ur(r_3)"] & r_1, r_2, r_4, r_5
\end{tikzcd}$$

Fig. 5. The transition graph computed in Example 4 using the incremental algorithm.

The goal is not reached. The transition graph $G(I)$ computed using the non-incremental algorithm is given in Figure 1.

Suppose that rule $can\_revoke(r_3)$ is added to the policy. Our incremental algorithm works as follows. First, the algorithm processes the initial state $\{r_1, r_2\}$. Because r_3 is not in the initial state, no new transitions are added. The algorithm then processes state $\{r_1, r_2, r_3, r_4\}$ and adds a new transition $\{r_1, r_2, r_3, r_4\} \overset{ur(r_3)}{\to} \{r_1, r_2, r_4, r_5\}$. Because state $\{r_1, r_2, r_4, r_5\}$ contains the goal, the algorithm returns true. Figure 5 gives the resulting graph. □

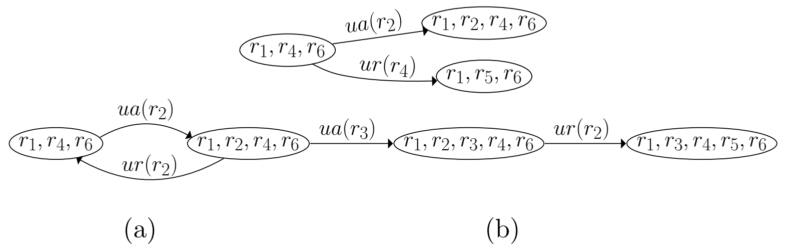


Fig. 6. Graphs computed in Example 5 using: (a) the non-incremental forward algorithm; (b) the incremental forward algorithm.

Deleting a can_revoke rule Suppose that $can\_revoke(T)$ is removed from ψ. Let $I_2 = \langle R_{0t}, \psi \setminus \{can\_revoke(T)\}, goal\rangle$. If T is neither a relevant mixed role nor a negative role in ϕ, then $can\_revoke(T)$ is irrelevant to the goal and hence deleting $can\_revoke(T)$ does not change the analysis result. Otherwise, the initial state may change and transitions that revoke T should be deleted. Figure 7 gives the pseudocode of the algorithm.

Since only *can_assign* rules are used during slicing, deleting a *can_revoke* rule does not change the result of the slicing transformation. If T is a mixed role relevant to the goal, the algorithm starts from the initial state of $G(I)$ and deletes transitions that revoke T (lines 3-14). If T is in R_{0t} and is both negative and non-positive, then after deleting $can\_revoke(T)$, T should be added back to the initial state (lines 15-32). In this case, the algorithm computes a set R_T of roles that may be invalidated by T (function *computeRT* in line 16). A role $r \in R_T$ if (1) T is a negative precondition of a *can_assign* rule whose target role is r or (2) there exists a role $r' \in R_T$ such that r' is a positive precondition of a *can_assign* rule whose target role is r. If the number of relevant roles in R_T that are both positive and non-negative is small (≤ 3 in our implementation), then for every transition $s \overset{\alpha}{\rightarrow} s_1$ in $G(I)$ and state $s' \in G_{inc}(I_2)$ computed from s, the algorithm computes transition $s' \overset{\alpha}{\rightarrow} s'_1$ by removing roles in R_T that do not appear in s' from s_1 and computing the closure of the resulting state (lines 17–24). The algorithm also checks if transitions that add mixed roles in R_T are invalidated, and if so, removes the transitions (line 22). Otherwise, the algorithm performs non-incremental analysis as removing a large number of roles from every state may be more expensive than a complete re-analysis (line 33). If the state containing the goal is deleted, the algorithm performs non-incremental analysis from states marked as "UnProcessed" (lines 34-35).

Example 5. *Consider the analysis problem instance in Example 2.*

The goal is reachable. The transition graph computed using the non-incremental algorithm is given in Figure 6(a).

Assume that $can\_revoke(r_4)$ is deleted. Because r_4 is a mixed role, deleting this rule may affect the analysis result. Our algorithm starts from the initial state $\{r_1, r_4, r_6\}$ and updates the graph. The initial state is still valid, but the transition $\{r_1, r_4, r_6\} \overset{ur(r_4)}{\rightarrow} \{r_1, r_5, r_6\}$ is not valid any more and hence is removed. As a result, the goal state $\{r_1, r_5, r_6\}$ is removed. Next, the algorithm performs non-incremental analysis from state $\{r_1, r_2, r_4, r_6\}$, and adds the following three transitions to the graph: $\{r_1, r_2, r_4, r_6\} \overset{ua(r_3)}{\rightarrow} \{r_1, r_2, r_3, r_4, r_6\}$, $\{r_1, r_2, r_4, r_6\} \overset{ur(r_2)}{\rightarrow} \{r_1, r_4, r_6\}$, and

```
1  $init' = closure(InitRm_{I_2}(R_{0t}), I_2)$
2  if $goal \subseteq init'$ then $add\ init'\ to\ G_{inc}(I_2)$;  return $true$ endif
3  if $T \in Rel_+(I_2) \cap Rel_-(I_2)$ then
4    $W = reached = \{init\}$
5    while $W \neq \emptyset$ do
6      $Remove\ s\ from\ W$
7      for $s \xrightarrow{\alpha} s_1 \in G(I)$ do
8        if $\alpha \neq ur(T)$ then
9          $add\ s \xrightarrow{\alpha} s_1\ to\ G_{inc}(I_2)$
10         if $goal \subseteq s_1$ then $markUnproc(W \cup \{s\})$;  return $true$ endif
11         if $s_1 \notin reached$ then $W = W \cup \{s_1\}$;  $reached = reached \cup \{s_1\}$ endif
12       elseif $goal \subseteq s_1$ then $markUnproc(\{s\})$ endif
13     endfor
14   endwhile
15 elseif $T \in (R_{0t} \cap Rel_-(I_2))$ then
16   $R_T = computeRT(T, I_2)$
17   if $|R_T \cap (Rel_+(I_2) \setminus Rel_-(I_2))|\ is\ small$ then
18     $W_1 = reached_1 = \{\langle init, init' \rangle\}$
19     while $W_1 \neq \emptyset$ do
20       $Remove\ \langle s, s' \rangle\ from\ W_1$
21       for $s \xrightarrow{\alpha} s_1 \in G(I)$ do
22         if $\alpha\ is\ enabled\ from\ s'$ then
23           $s'_1 = closure((s_1 \setminus (((R_T \cap (Rel_+(I_2) \setminus Rel_-(I_2))) \setminus s') \cup \{T\}), I_2))$
24           $add\ s' \xrightarrow{\alpha} s'_1\ to\ G_{inc}(I_2)$
25           if $goal \subseteq s'_1$ then $markUnproc(\{s'_i | \langle s_i, s'_i \rangle \in W_1\} \cup \{s'\})$;  return $true$ endif
26           if $\langle s_1, s'_1 \rangle \notin reached_1$ then $W_1 = W_1 \cup \{\langle s_1, s'_1 \rangle\}$;
27             $reached_1 = reached_1 \cup \{\langle s_1, s'_1 \rangle\}$
28           endif
29         elseif $goal \subseteq s_1$ then $markUnproc(\{s'\})$ endif
30       endfor
31     endwhile
32   else $perform\ non-incremental\ analysis\ from\ init'$ endif
33 else $return\ the\ analysis\ result\ of\ I$ endif
34 if $G_{inc}(I_2) == \emptyset$ then $add\ init'\ to\ G_{inc}(I_2)$; return $false$ endif
35 $process\ all\ states\ marked\ UnProcessed\ with\ non-incremental\ alg.$
```

Fig. 7. Pseudocode for deleting *can_revoke*(T)

$\{r_1, r_2, r_3, r_4, r_6\} \xrightarrow{ur(r_2)} \{r_1, r_3, r_4, r_5, r_6\}$. *Because state* $\{r_1, r_3, r_4, r_5, r_6\}$ *contains the goal, the algorithm returns true. The resulting graph is given in Figure 6(b).* □

Adding a can_assign rule Suppose that $can\_assign(P \wedge \neg N, T)$ is added to the policy ψ. Let $I_3 = \langle R_{0t}, \psi \cup \{can\_assign(P \wedge \neg N, T)\}, goal \rangle$. Figure 8 gives the pseudocode for constructing graph $G_{inc}(I_3)$ from $G(I)$.

If $T \notin Rel_+(I)$, then $can\_assign(P \wedge \neg N, T)$ is not relevant to the goal (i.e., $can\_assign(P \wedge \neg N, T) \notin RelRule$). As a result, adding $can\_assign(P \wedge \neg N, T)$ does not enable any new transitions and does not disable any existing transitions in the graph, and hence does not affect the transition graph and the analysis result (line 1). Otherwise, the classification of roles may change: (1) An irrelevant role in I may become relevant in I_3; (2) A relevant role that is both positive and non-negative in I may become a mixed role in I_3; and (3) A relevant role that is both negative and non-positive in I may become a mixed role in I_3. In this case, the algorithm

```
if $T \notin Rel_+(I)$ then $return\ the\ analysis\ result\ of\ I$
else
  $\langle Rel_+(I_3), Rel_-(I_3)\rangle = inc\_slicing()$
  $init' = closure(InitRm_{I_3}(R_{0t}), I_3)$
  if $goal \subseteq init'$ then $add\ init'\ to\ G_{inc}(I_3)$; return $true$ endif
  $RevRoles = ((init' \setminus init) \cap (Rel_+(I_3) \cap Rel_-(I_3)) \setminus Irrev)$
  $PosNonnegToMix = ((Rel_+(I) \setminus Rel_-(I)) \cap (Rel_+(I_3) \cap Rel_-(I_3)))$
  $AddRoles = init \cap PosNonnegToMix$
  if $AddRoles \cup RevRoles \neq \emptyset$ then
    $\langle answer, lastState\rangle = addTransSeq(init, init', RevRoles, AddRoles)$
    if $answer == true$ then return $true$ else $W = reached = \{\langle init, lastState\rangle\}$ endif
  else $W = reached = \{\langle init, init'\rangle\}$ endif
  while $W \neq \emptyset$ do
    $Remove\ \langle s, s'\rangle\ from\ W$
    for $s \overset{\alpha}{\to} s_1 \in G(I)$ do
      $AddRoles = s_1 \cap PosNonnegToMix$
      if $AddRoles \neq \emptyset$ then
        $\langle answer, s'_1\rangle = addTransSeq(s, s', \emptyset, AddRoles)$
        if $answer == true$ then $markUnproc(\{s'_j | \langle s_j, s'_j\rangle \in W\} \cup \{s'\})$; return $true$ endif
      else
        $s'_1 = closure(s_1 \cup (s' \setminus s), I_3)$
        $add\ s' \overset{\alpha}{\to} s'_1\ to\ G_{inc}(I_3)$
        if $goal \subseteq s'_1$ then $markUnproc(\{s'_j | \langle s_j, s'_j\rangle \in W\} \cup \{s'\})$; return $true$ endif
      endif
      if $\langle s_1, s'_1\rangle \notin reached$ then $reached = reached \cup \{\langle s_1, s'_1\rangle\}$; $W = W \cup \{\langle s_1, s'_1\rangle\}$ endif
    endfor
    if $addNewTrans(s') == true$ then $markUnproc(\{s'_j | \langle s_j, s'_j\rangle \in W\} \cup \{s'\})$; return $true$ endif
  endwhile
  if $G_{inc}(I_3) == \emptyset$ then $add\ init'\ to\ G_{inc}(I_3)$; return $false$ endif
  $process\ all\ states\ marked\ UnProcessed\ with\ non-incremental\ alg.$
endif
```

Fig. 8. Pseudocode for adding $can\_assign(P \wedge \neg N, T)$

performs incremental slicing to compute $Rel_+(I_3)$ and $Rel_-(I_3)$ (function $inc\_slicing$ in line 3): $Rel_+(I_3) = Rel_+(I) \cup \{r | r$ is a positive role relevant to $T\}$ and $Rel_-(I_3) = Rel_-(I) \cup \{r | r$ is a negative role relevant to $T\}$. The algorithm also computes a set $IncRelRule$ of rules, which are sufficient to consider during the incremental analysis. Let ρ be a can_assign rule, $target(\rho)$ be the target role of ρ, and $poscond(\rho)$ be the set of positive preconditions of ρ. $IncRelRule$ is defined as follows:

(1) $can\_assign(P \wedge \neg N, T) \in IncRelRule$.

(2) a can_assign rule $\rho \in IncRelRule$ if

  (a) $target(\rho) \in Rel_+(I_3)$ and there exists $\rho' \in IncRelRule$ such that $target(\rho') \in poscond(\rho)$ or

  (b) $target(\rho)$ is a positive role relevant to T.

(3) $can\_revoke(r) \in IncRelRule$ if r is a mixed role in I_3 or is a negative role in R_{0t}.

$IncRelRule$ consists of (1) the new rule, (2a) relevant can_assign rules enabled by the new rule, (2b) can_assign rules that enable the new rule, and (3) can_revoke rules which revoke mixed roles or negative roles in R_{0t}.

Next, the algorithm computes the new initial state $init'$ = closure($InitRm_{I_3}(R_{0t}), I_3$) (line 4), which may be different from the initial state

$init$ = closure($InitRm_I(R_{0t}), I$) of $G(I)$. Theorem 1 gives the relationship between $init'$ and $init$, which enables us to reuse $G(I)$ to construct $G_{inc}(I_3)$ incrementally.

Case (1) in Theorem 1 states that the initial state does not change after the rule is added. In Case (2), new roles are added to the initial state, but no roles are turned from both positive and non-negative to mixed. In these two cases, the algorithm adds roles in $init' \setminus init$ to $init$ and updates the graph (lines 20–24). In case (3), some roles in $init$ are turned from both positive and non-negative to mixed ($AddRoles$ in line 8), from irrelevant to relevant, or from revocable non-positive to mixed ($RevRoles$ in line 6). In this case, the algorithm calls function *addTransSeq* (line 10) to add a sequence of transitions from $init'$ to closure($init \cup s_{m-1}$, I_3), which revokes roles in $RevRoles$, adds roles in $AddRoles$, and marks new states not containing the goal as "UnProcessed". This function returns $\langle answer, lastState\rangle$ where $answer$ is true if the sequence contains the goal state and is false otherwise, and $lastState$ is the last state of the sequence.

Finally, the algorithm calls function *addNewTrans* to add new transitions from $init'$ (line 27) using rules in $IncRelRule$ and marks new states as "UnProcessed". The above process is then repeated on states reachable from $init$.

Theorem 1. *Let* $I = \langle R_{0t}, \psi, goal\rangle$ *and* $I_3 = \langle R_{0t}, \psi \cup \{can\_assign(P \wedge \neg N, T)\}, goal\rangle$. *Also, let* $Rev(I)$ *be a set of revocable roles in* I, $init$ = closure($InitRm_I(R_{0t}), I$), $init'$ = closure($InitRm_{I_3}(R_{0t}), I_3$). *One of the following holds:*

- *(1)* $init = init'$;
- *(2)* $init'$ = closure($init, I_3$); *or*
- *(3)* $G(I_3)$ *contains the following sequence of transitions:* $init' \overset{ur(r_1)}{\rightarrow} s_1 \ldots \overset{ur(r_n)}{\rightarrow} s_n \overset{ua(r_{n+1})}{\rightarrow} s_{n+1} \ldots s_{m-1} \overset{ua(r_m)}{\rightarrow}$ closure($init \cup s_{m-1}, I_3$) *where*
  - $\{r_1, \ldots, r_n\} = ((Rel_+(I_3) \cap Rel_-(I_3)) \cap (init' \setminus init) \cap Rev(I))$ *(i.e.,* $r_1, \ldots, r_n$ *are revocable mixed roles in* $init' \setminus init$*),*
  - $\{r_{n+1}, \ldots, r_m\} = ((Rel_+(I) \setminus Rel_-(I)) \cap Rel_-(I_3) \cap (init \setminus init'))$ *(i.e.* $r_{n+1}, \ldots, r_m$ *are roles in* $init \setminus init'$ *that are turned from both positive and non-negative to mixed).*

Proof: The proof is given below.
Case (1): $init = init'$ if all of the following conditions hold:

- $(R_{0t} \setminus (Rel_+(I) \cup Rel_-(I)) \cap (Rel_+(I_3) \cup Rel_-(I_3)) = \emptyset$ (i.e. no irrelevant roles in R_{0t} become relevant after the can_assign rule is added),
- $((init \setminus R_{0t}) \cap (Rel_+(I) \setminus Rel_-(I)) \cap Rel_-(I_3) = \emptyset$ (i.e. no relevant roles in $init \setminus R_{0t}$ that are both positive and non-negative become mixed roles after the can_assign rule is added),
- $(R_{0t} \cap Rev(I) \cap (Rel_-(I) \setminus Rel_+(I))) \cap Rel_+(I_3) = \emptyset$ (i.e., no revocable roles in R_{0t} that are both negative and non-positive become mixed roles after the can_assign rule is added), and
- $can\_assign(P \wedge \neg N, T)$ does not enable new invisible transitions from $init$.

Case (2): $init'$ can be computed by executing all invisible transitions enabled in I_3 from $init$ (i.e. $init' = closure(init, I_3)$) if all of the following conditions hold:

- $(R_{0t} \setminus (Rel_+(I) \cup Rel_-(I)) \cap (Rel_+(I_3) \cup Rel_-(I_3)) = \emptyset$,
- $((init \setminus R_{0t}) \cap (Rel_+(I) \setminus Rel_-(I)) \cap Rel_-(I_3) = \emptyset$,
- $(R_{0t} \cap Rev(I) \cap (Rel_-(I) \setminus Rel_+(I))) \cap Rel_+(I_3) = \emptyset$, and
- $can\_assign(P \wedge \neg N, T)$ enables new invisible transitions from $init$.

Case (3): Below, we show that if the above conditions do not hold, (3) holds. We first prove the following lemma and then use this lemma to prove the theorem.

Lemma 1. *Let $I = \langle R_{0t}, \psi, goal\rangle$ and $I_3 = \langle R_{0t}, \psi \cup \{can\_assign(P \wedge \neg N, T)\}, goal\rangle$. Also, let $init = \mathrm{closure}(InitRm_I(R_{0t}), I)$ and $init' = \mathrm{closure}(InitRm_{I_3}(R_{0t}), I_3)$. If $init' \setminus init$ does not contain revocable mixed roles, then init is reachable from $init'$ through the following sequence of transitions: $init' \overset{ua(r_1)}{\rightarrow} s_1 \overset{ua(r_2)}{\rightarrow} \ldots \overset{ua(r_m)}{\rightarrow} closure(init \cup s_{m-1}, I_3)$, where $\{r_1, \ldots, r_m\} = ((Rel_+(I) \setminus Rel_-(I)) \cap Rel_-(I_3) \cap (init \setminus init'))$.*

Proof: We prove the lemma by induction on the length of the sequence of transitions. *Base Case:* If $init \setminus init'$ does not contain mixed roles, then either case (1) or case (2) applies.
Induction: Assume that the lemma holds for $m = k - 1$. We prove the lemma for $m = k$.

Assume that $\{r_1, \ldots, r_k\} = ((Rel_+(I) \setminus Rel_-(I)) \cap Rel_-(I_3) \cap (init \setminus init'))$, i.e. $\{r_1, \ldots, r_k\}$ are roles in $init \setminus init'$ that are turned from both positive and nonnegative to mixed after the can_assign rule is added, and were added to $init$ through invisible transitions. Because adding a can_assign rule does not disable transitions, we can perform transition $init' \overset{ua(r_1)}{\rightarrow} s_1$. Since this transition adds only r_1 and non-mixed roles to s_1, $\{r_2, \ldots, r_k\}$ are roles in $init \setminus s_1$ that are turned from both positive and nonnegative to mixed after the can_assign rule is added. By induction hypothesis, $init$ is reachable from s_1 through the sequence of transitions $s_1 \overset{ua(r_1)}{\rightarrow} \ldots \overset{ua(r_k)}{\rightarrow}$ closure($init \cup s_{k-1}$). Thus, the lemma holds.

End-of-lemma □

Below, we prove the theorem by induction on n.

Base Case: If $init' \setminus init$ does not contain roles that are turned from irrelevant to revocable mixed roles (i.e., $n = 0$), then Lemma 1 applies.
Induction: Assume that (3) holds for $n = k - 1$. We prove (3) for $n = k$.

Assume that $\{r_1, \ldots, r_k\}$ are revocable mixed roles in $init' \setminus init$. Since adding can_assign rule does not disable any transitions, we can perform transition $init' \overset{ur(r_1)}{\rightarrow} s_1$. This transition revokes only role r_1 from $init'$ and adds only non-mixed roles. As a result, $s_1 \setminus init$ contains mixed revocable roles $\{r_2, \ldots, r_k\}$. By induction hypothesis, $init$ is reachable from s_1 through a sequence of transitions $s_1 \overset{ur(r_2)}{\rightarrow} \ldots \overset{ur(r_k)}{\rightarrow} closure(init \cup s_{k-1}, I_3)$. Thus, case (3) of Theorem 1 holds. □

Example 6. *Consider the ARBAC policy ψ in Example 1 and the reachability analysis problem instance $I = \langle\emptyset, \psi, \{r_5\}\rangle$.*

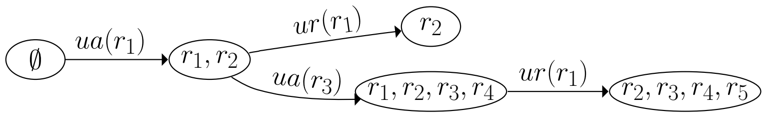


Fig. 9. The transition graph computed in Example 6 using the incremental algorithm.

The transition graph computed using the non-incremental algorithm is given in Figure 1. The goal is not reachable.

Suppose that $can\_assign(r_3 \wedge \neg r_1, r_5)$ is added to the policy. Our incremental algorithm works as follows. First, the algorithm performs incremental slicing: role r_1 is turned from both positive and non-negative to mixed. IncRelRule consists of rules 1-3, $can\_assign(r_3 \wedge \neg r_1, r_5)$, and $can\_revoke(r_1)$. The new initial state is $\emptyset$. Next, the algorithm applies case (3) of Theorem 1, which adds transition $\emptyset \overset{ua(r_1)}{\to} \{r_1, r_2\}$ to the graph. The algorithm also adds transition $\{r_1, r_2\} \overset{ur(r_1)}{\to} \{r_2\}$ to the graph and marks states $\emptyset$ and $\{r_2\}$ as "UnProcessed". The algorithm then processes transition $\{r_1, r_2\} \overset{ua(r_3)}{\to} \{r_1, r_2, r_3, r_4\}$. This transition is still valid and hence remains in the graph. Finally, the algorithm computes new transitions from state $\{r_1, r_2, r_3, r_4\}$ and adds transition $\{r_1, r_2, r_3, r_4\} \overset{ur(r_1)}{\to} \{r_2, r_3, r_4, r_5\}$ to the graph. Because state $\{r_2, r_3, r_4, r_5\}$ contains the goal, the algorithm returns true. The resulting graph is given in Figure 9. $\square$

Deleting a can_assign rule Suppose that $can\_assign(P \wedge \neg N, T)$ is deleted from policy ψ. Let $I_4 = \langle R_{0t}, \psi \setminus \{can\_assign(P \wedge \neg N, T)\}, goal \rangle$. If T is not a positive role relevant to the goal, then $can\_assign(P \wedge \neg N, T)$ is not relevant to the goal (i.e., *RelRule* does not contain $can\_assign(P \wedge \neg N, T)$) and hence $can\_assign(P \wedge \neg N, T)$ is not applied in the previous analysis. As a result, deleting this rule does not affect the transition graph. Otherwise, role T and roles that are reachable through T may become unreachable. Let A_T be a set of roles that may be reachable through T. A role r is in A_T if: (1) T is a positive precondition of a *can_assign* rule whose target role is r or (2) there exists a role $r' \in A_T$ such that r' is a positive precondition of a *can_assign* rule whose target role is r.

Deleting $can\_assign(P \wedge \neg N, T)$ may change the classification of T from mixed to both positive and non-negative. This occurs when targets of all *can_assign* rules, which contain T in their negative preconditions, become non-positive after the rule is deleted. Similarly, roles other than T may change from mixed to both positive and non-negative (*MixtoNonneg*), from mixed to both negative and non-positive (*MixtoNonpos*), and from relevant to irrelevant (*RevtoIrr*). Figure 10 gives the pseudocode for deleting a *can_assign* rule.

First, the algorithm performs slicing transformation (function *slicing* in line 2) and computes the new initial state $init' = closure(InitRm_{I_4}(R_{0t}), I_4)$ (line 3). $init'$ may be different from the initial state $init = closure(InitRm_I(R_{0t}), I)$ of $G(I)$. Theorem 2 gives the relationship between *init* and $init'$, which enables us to reuse $G(I)$ to construct $G_{inc}(I_4)$.

Case (1) states that the initial state does not change. In Case (2), *init* does not contain roles turned from mixed to both positive and non-negative, but may contain roles turned from relevant to irrelevant, revocable roles turned from mixed to non-positive, or roles that cannot be re-derived after the *can_assign* rule is deleted. In this case, the algorithm updates the graph from *init*, removes such roles from *init* and states reachable from *init*, and removes transitions that add or revoke such roles. In case (3), *init* contains roles turned from mixed to both positive and non-negative. In this case, the algorithm identifies the state $(init' \cup (s_{n-1} \cap (RevtoIrr \cup Invalid \cup MixtoNonpos)))$ in $G(I)$, and updates the graph by removing roles in $(s_{n-1} \cap (RevtoIrr \cup Invalid \cup MixtoNonpos))$ from this state and all states reachable from this state.

The graph is updated as follows. First, the algorithm computes set A_T (function *computeAT* in line 8). Next, for every transition $s \xrightarrow{\alpha} s_1$ in $G(I)$, if α adds/revokes a role that is turned from mixed to non-positive or if α can no longer be derived, the algorithm removes the transition (lines 17-19). Otherwise, if $s_1 \setminus s$ contains T and T cannot be re-derived, the algorithm removes T and all roles that cannot be derived without T from s_1 (lines 21-22). If α adds a role that is turned from mixed to both positive and non-negative, the algorithm removes the transition and updates the graph using a way similar to case (3) of Theorem 2 (function *findTransSeq* in line 28). *findTransSeq* returns the last state of the sequence in case (3) of Theorem 2.

Theorem 2. *Let* $I = \langle R_{0t}, \psi, goal\rangle$, $I_4 = \langle R_{0t}, \psi \setminus \{can\_assign(P \wedge \neg N, T)\}, goal\rangle$, $init = closure(InitRm_I(R_{0t}), I)$, *and* $init' = closure(InitRm_{I_4}(R_{0t}), I_4)$. *Also, let* $Rev(I)$ *be a set of revocable roles in* I, $RevtoIrr = (Rel_+(I) \cup Rel_-(I)) \setminus (Rel_+(I_4) \cup Rel_-(I_4))$, $MixtoNonpos = (Rel_+(I) \cap Rel_-(I)) \setminus Rel_+(I_4)$, $MixtoNonneg = (Rel_+(I) \cap Rel_-(I)) \setminus Rel_-(I_4)$, $Invalid = (init \setminus init') \cap (A_T \cup \{T\})$. *One of the following holds:*

- *(1)* $init' = init$*;*
- *(2)* $init' = init \setminus (RevtoIrr \cup Invalid \cup \{S \in MixtoNonpos | S \text{ is revocable}\})$*;*
- *(3)* $G(I)$ *contains the following sequence of transitions:* $init \xrightarrow{ua(r_1)} s_1 \ldots s_{n-1} \xrightarrow{ua(r_n)} (init' \cup (s_{n-1} \cap (RevtoIrr \cup Invalid \cup MixtoNonpos)))$ *where* $\{r_1, \ldots, r_n\} = (init' \setminus init) \cap MixtoNonneg$.

Proof: Below, we prove each case of the theorem.

Case (1): (1) holds if all of the following conditions hold:

- $((Rel_+(I) \cup Rel_-(I)) \cap R_{0t}) = ((Rel_+(I_4) \cup Rel_-(I_4)) \cap R_{0t})$ (i.e. all relevant roles in R_{0t} are still relevant after the *can_assign* rule is deleted),
- $Rel_+(I) \cap Rel_-(I) \cap R_{0t} \cap Rev(I)) \cap (Rel_-(I_4) \setminus Rel_+(I_4)) = \emptyset$ (i.e. none of the revocable mixed roles in R_{0t} become non-positive after the *can_assign* rule is deleted),
- there does not exist $init \xrightarrow{ua(r)} S$ such that $r \in (Rel_+(I_4) \setminus Rel_-(I_4))$, and
- deleting the *can_assign* rule does not disable any invisible transition in $G(I)$ that adds roles to *init*.

Case (2): (2) holds if there does not exist $init \xrightarrow{ua(r)}$ such that $r \in (Rel_+(I_4) \setminus Rel_-(I_4))$, and one of the following conditions hold:

- $((Rel_+(I) \cup Rel_-(I)) \cap R_{0t}) \setminus ((Rel_+(I_4) \cup Rel_-(I_4)) \cap R_{0t}) \neq \emptyset$ (i.e. at least one relevant role in R_{0t} becomes irrelevant after the *can_assign* rule is deleted),
- $((Rel_+(I) \cup Rel_-(I)) \cap R_{0t}) \cap ((Rel_+(I_4) \setminus Rel_-(I_4)) \neq \emptyset$ (i.e. at least one revocable mixed role in R_{0t} becomes non-positive after the *can_assign* rule is deleted),
- deleting the *can_assign* rule disables at least one of the invisible transitions in $G(I)$ that add roles to *init*.

Case (3): Below, we show that, if the above conditions do not hold, (3) holds. The proof is by induction on the length of the sequence of transitions.

Base Case: If $init' \setminus init$ does not contain roles that are both positive and non-negative, then either case (1) or case (2) apply.

Induction: Assume that case (3) of Theorem 2 holds for $n = k - 1$. Below, we prove case (3) for $n = k$.

Let $\{r_1, \ldots, r_k\} = (init' \setminus init) \cap MixtoNonneg$, i.e. $\{r_1, \ldots, r_k\}$ are roles in $init' \setminus init$ that are turned from mixed to both positive and non-negative. such roles were added to *init* through visible transitions. From *init*, there is a transition $init \overset{ua(r_1)}{\rightarrow} s_1$, which is enabled without the deleted *can_assign* rule. This transition only adds r_1 and non-mixed roles to s_1. As a result, $\{r_3, \ldots, r_k\} = (init' \setminus s_1) \cap MixtoNonneg$. By induction hypothesis, $init'$ is reachable from s_1 through a sequence of transitions $s_1 \overset{ua(r_2)}{\rightarrow} \ldots s_{k-1} \overset{ua(r_k)}{\rightarrow} (init' \cup (s_{k-1} \cap (RevtoIrr \cup Invalid \cup MixtoNonpos)))$. Thus, (3) holds. □

Example 7. *Consider the analysis problem instance in Example 2. The goal is reachable. The transition graph computed using the non-incremental algorithm is given in Figure 6(a).*

Suppose that rule 6 is removed from the policy. First, the algorithm computes $RevtoIrr = \{r_4, r_6\}$, $MixtoNonneg = \{r_3\}$, *and* $MixtoNonpos = \emptyset$. *Next, the algorithm computes the new initial state* $\{r_1\}$ *by removing* r_4 *and* r_6 *from the previous initial state. The algorithm then processes transition* $\{r_1, r_4, r_6\} \overset{ua(r_2)}{\rightarrow} \{r_1, r_2, r_4, r_6\}$*: it removes* r_4 *and* r_6 *from both states, and adds* r_3 *to the target state. Finally, the algorithm removes transition* $\{r_1, r_4, r_6\} \overset{ur(r_4)}{\rightarrow} \{r_1, r_5, r_6\}$*, because* r_4 *becomes irrelevant. As a result, the goal state* $\{r_1, r_5, r_6\}$ *is invalidated. The algorithm then computes transitions from state* $\{r_1, r_2, r_3\}$*, which is marked "UnProcessed", using the non-incremental algorithm. This results in a new transition* $\{r_1, r_2, r_3\} \overset{ur(r_2)}{\rightarrow} \{r_1, r_3, r_5\}$. *Because* $\{r_1, r_3, r_5\}$ *contains the goal, the algorithm returns true. The resulting graph is given in Figure 11.* □

3.4 Lazy Incremental Forward Algorithm

This section presents a lazy incremental analysis algorithm that delays updates to the transition graph until an operation, which may affect the analysis result, is performed. Our lazy analysis algorithm can be used to handle a sequence of changes on ARBAC policies. In this section, we present only the algorithm for the case where the analysis result of the original policy is true. The case where the analysis result is false is handled similarly. Let $I = \langle R_{0t}, \psi, goal \rangle$ be a reachability analysis problem instance. The algorithm is described below.

```
if $T \notin Rel_+(I)$ then $return\ the\ analysis\ result\ of\ I$ endif
$\langle Rel_+(I_4), Rel_-(I_4)\rangle = slicing()$
$init' = closure(InitRm_{I_4}(R_{0t}), I_4)$
if $goal \subseteq init'$ then $add\ init'\ to\ G_{inc}(I_4)$; return $true$ endif
$MixtoNonneg = (Rel_+(I) \cap Rel_-(I)) \cap (Rel_+(I_4) \setminus Rel_-(I_4))$
$MixtoNonpos = \{r | r \in ((Rel_+(I) \cap Rel_-(I)) \cap (Rel_-(I_4) \setminus Rel_+(I_4)))\ and\ can\_revoke(r) \in \psi\}$
$RevtoIrr = ((Rel_+(I) \cup Rel_-(I)) \setminus (Rel_+(I_4) \cup Rel_-(I_4)))$
$A_T = computeAT(T, I_4)$
if $(init' \cap MixtoNonNeg \neq \emptyset)$ then
  $Invalid = ((init \setminus init') \cap (A_T \cup \{T\}))$
  $lastState = findTransSeq(init, init', Invalid, MixtoNonneg, MixtoNonpos, RevtoIrr)$
  $W = \{\langle lastState, init'\rangle\}$
else $W = \{\langle init, init'\rangle\}$ endif
while $W \neq$ do
  $Remove\ \langle s, s'\rangle\ from\ W$
  for $s \overset{\alpha}{\to} s_1 \in G(I)$ do
    if $(\alpha = ua(r)/ur(r)\ where\ r \in (MixtoNonpos \cup MixtoNonneg))$
      or $(\alpha\ is\ not\ enabled\ from\ s')$ then
      if $goal \subseteq s_1$ then $markUnproc(\{s'\})$ endif
    else
      if $(\alpha \neq ua(T))$ and $(T \in (s_1 \setminus s'))$ then
        $s'_1 = closure(s_1 \setminus (((A_T \cup \{T\}) \setminus s') \cup (RevtoIrr \cup MixtoNonpos)), I_4)$
      else $s'_1 = closure(s_1 \setminus (RevtoIrr \cup MixtoNonpos), I_4)$ endif
      $add\ s' \overset{\alpha}{\to} s'_1\ to\ G_{inc}(I_4)$
      if $goal \subseteq s'_1$ then $markUnproc(\{s'_i | \langle s_i, s'_i\rangle \in W\} \cup \{s'\})$; return $true$ endif
      if $s'_1 \cap MixtoNonneg \neq \emptyset$ then
        $Invalid = ((s_1 \setminus s'_1) \cap (A_T \cup \{T\}))$
        $lastState = findTransSeq(s_1, s'_1, Invalid, MixtoNonneg, MixtoNonpos, RevtoIrr)$
      else $lastState = s_1$ endif
      if $\langle lastState, s'_1\rangle \notin\ reached$ then
        $W = W \cup \{\langle lastState, s'_1\rangle\}$; $reached = reached \cup \{\langle lastState, s'_1\rangle\}$
      endif
    endif
  endfor
endwhile
if $G_{inc}(I_4) == \emptyset$ then $add\ init'\ to\ G_{inc}(I_4)$; return $false$ endif
$process\ all\ states\ marked\ UnProcessed\ with\ non-incremental\ alg.$
```

Fig. 10. Pseudocode for deleting $can\_assign(P \wedge \neg N, T)$

Adding a can_assign or a can_revoke rule If the analysis result is true, then adding a *can_assign* or a *can_revoke* rule does not affect the analysis result. This is because adding a rule does not disable any ARBAC rules applied in the previous analysis and hence the goal can still be reached without the added rule. However, adding a *can_assign* or a *can_revoke* rule may affect the transition graph since it may enable new transitions. In this case, we do not update the graph. Instead, we store the rule in a set *DelayedRule*. This set will be used to update the transition graph when an operation that may affect the analysis result is performed.

Deleting a can_assign or a can_revoke rule Assume that $can\_assign(P \wedge \neg N, T)$ is deleted from ψ. If T is not a positive role relevant to the goal, then $can\_assign(P \wedge \neg N, T)$ is irrelevant to the goal (i.e. $can\_assign(P \wedge \neg N, T) \notin RelRule$), and hence this rule was not applied during the previous analysis. As a result, deleting $can\_assign(P \wedge \neg N, T)$ does not affect the analysis result and we simply remove the

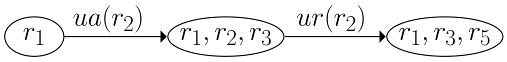


Fig. 11. The transition graph computed in Example 7 using the incremental algorithm.

rule from the policy. Otherwise, let $\psi' = (\psi \setminus \{can\_assign(P \wedge \neg N, T)\}) \cup DelayedRule$ and $I' = \langle R_{0t}, \psi', goal \rangle$. Our algorithm works as follows. First, we perform the slicing mechanism to compute a set of positive and negative relevant roles $Rel_+(I')$ and $Rel_-(I')$. We then update the transition graph using the deleted rule and delayed operations that may affect the analysis result after the rule is deleted. Such operations include the added *can_assign* rules in *DelayedRule* whose target roles are in $Rel_+(I')$, and the added *can_revoke* rules in *DelayedRule* that revoke relevant mixed roles or negative roles in R_{0t}. Finally, We update the graph using one of the following two approaches.

In the first approach, we update every state and transition of the graph by performing all operations of IncFwd2 described in Section 3.3 with the following difference: when applying Theorem 1, not all transitions adding roles $r_{n+1}, \ldots, r_m$ may be enabled in I' because some of them may depend on the deleted *can_assign* rule.

In the second approach, we consider the operation that deletes $can\_assign(P \wedge \neg N, T)$ and apply IncFwd2 for deleting *can_assign* to update the graph. If the analysis result changes from true to false, the algorithm updates the graph using rules in *DelayedRule*. The graph is updated in a way similar to algorithms in Figures 4 and 8, but considering multiple added rules.

The second approach is expected to perform better than the first one if the analysis result does not change after the rule is deleted, and worse otherwise. This is because in the former case, the graph is processed once, but in the latter case, the graph is processed twice. Both approaches are expected to perform better than IncFwd2 where each change is processed individually, because the graph will be processed fewer times when applying these two approaches. Our implementation adopts the first approach.

Deleting a *can_revoke* rule is handled similarly.

Example 8. *Consider the analysis problem instance in Example 2.*

The goal is reachable. The transition graph computed using the non-incremental algorithm is given in Figure 6 (a).

Assume that the following sequence of changes is made to the policy: (1) delete rule 7, (2) add $can\_assign(r_4 \wedge \neg r_5, r_1)$, (3) add $can\_assign(r_1, r_3)$, (4) delete $can\_revoke(r_6)$, and (5) delete rule 3. These operations are processed as follows. Because r_7 is a non-positive role, deleting rule 7 does not affect the analysis result. Rules $can\_assign(r_1, r_3)$ and $can\_assign(r_4 \wedge \neg r_5, r_1)$ do not change the analysis result and are added to DelayedRule. Because r_6 is neither a mixed role nor a negative role in R_{0t}, deleting $can\_revoke(r_6)$ does not affect the result. Finally, because r_3 is a positive role, deleting rule 3 may affect the analysis result and the algorithm performs the following steps.

First, the algorithm performs slicing: role r_5 is turned from both positive and non-negative to mixed, and role r_2 is turned from mixed to non-positive. Next, the algorithm updates the graph from the initial state of $G(I)$ using operations that may affect the analysis result, i.e., (2), (3), and (5). Because operation (5) changes the classification *of role r_2 from mixed to non-positive, transition $\{r_1, r_4, r_6\} \overset{ua(r_2)}{\rightarrow} \{r_1, r_2, r_4, r_6\}$ is deleted. The algorithm then processes transition $\{r_1, r_4, r_6\} \overset{ur(r_4)}{\rightarrow} \{r_1, r_5, r_6\}$. Because r_5 becomes mixed, the algorithm applies Case (3) of Theorem 1 and adds transitions $\{r_1, r_4, r_6\} \overset{ur(r_4)}{\rightarrow} \{r_1, r_6\}$ and $\{r_1, r_6\} \overset{ua(r_5)}{\rightarrow} \{r_1, r_5, r_6\}$ to the graph. Since role r_5 is in the graph, the goal is reachable.*

$\square$

3.5 Incremental Backward Algorithm

This section presents a backward algorithm for incremental user-role reachability analysis of ARBAC. Similar to IncFwd2, our backward algorithm uses the graph G_b and the airs computed in the previous analysis to incrementally update the result. Ideas used in IncFwd1 and LazyInc are also applicable to the backward algorithm.

To support efficient incremental analysis, we extend the non-incremental algorithm as follows: (1) Prior to analysis, we compute a set of roles relevant to the goal, which enables us to quickly determine if a change to the policy may affect the analysis result. (2) We store the graph as well as the airs computed in a file. The initial nodes are also stored, in the order in which they are processed in the second stage. (3) For every node V, we associate every set of $airs(V)$ with the edge along which the set is computed. This enables us to quickly identify the set of airs computed from a given edge. (4) If an edge is processed in the second stage of the algorithm, the edge is marked 1; otherwise, the edge is marked 0. Let $airs'(V)$ denote the airs of node V computed by the incremental algorithm. Below, we describe the algorithm.

Adding a can_revoke rule Assume that *can_revoke*(T) is added to the policy. Since graph G_b is constructed using only *can_assign* rules, G_b is unaffected and we simply update the airs of nodes from the first initial node V_0. Let $rm(T, airs(V))$ denote $\{S \setminus \{T\} | S \in airs(V)\}$. $airs'(V)$ is computed as follows:

- $airs'(V_0) = rm(T, airs(V_0))$
- For every edge $V_1 \overset{\langle P \wedge \neg N, r \rangle}{\rightarrow} V_2$, $airs'(V_2)$ is computed as the union of the following sets:
  (1) $rm(T, airs(V_2))$
  (2) $\{S \cup (Irrev \cap (V_2 \setminus V_1)) | S \in airs'(V_1) \setminus rm(T, airs(V_1)),$
  $((Irrev \cap V_1) \cup S) \cap N = \emptyset\}$
  (3) $((T \in N) ? \{(S \setminus \{T\}) \cup (Irrev \cap (V_2 \setminus V_1)) | S \in airs(V_1),$
  $T \in S, ((Irrev \cap V_1) \cup (S \setminus \{T\})) \cap N = \emptyset\} : \emptyset)$

(1) contains $airs(V_2)$ with T removed from every set. (2) contains sets of additional irrevocable roles computed from new sets in airs of V_1 along the edge. (3) is computed from sets in $airs(V_1)$ that did not satisfy the negative precondition of the edge because they contained T; since T becomes revocable, they are added to $airs'(V_2)$.

If the goal is not reachable from V_0, we pick up the second initial node and repeat the above process. If the goal is reachable, the algorithm updates the airs of nodes until it encounters an edge marked 0. This is because, after a *can_revoke* rule is added, the goal that was previously unreachable may become reachable and the goal that was previously reachable may be reached earlier.

Example 9. *Consider the analysis problem instance in Example 1.*

The goal is not reachable. Figure 12 gives the graph constructed in the first stage.

There are two initial nodes: $\emptyset$ and $\{r_1\}$. $airs(\emptyset) = airs(\{r_1\}) = airs(\{r_2\}) = \{\emptyset\}$, $airs(\{r_3\}) = \{\{r_2\}\}$, $airs(\{r_4\}) = \{\{r_2, r_3\}\}$, and $airs(\{r_5\}) = \emptyset$.

Suppose that rule can_revoke(r_3) is added to the policy. The incremental algorithm starts from the initial node $\emptyset$ and updates the airs of nodes. Because $airs(\emptyset)$ does not contain r_3, $airs'(\emptyset) = \{\emptyset\}$. The algorithm then updates the airs of nodes reachable from $\emptyset$. Because the airs of nodes $\{r_1\}$, $\{r_2\}$, and $\{r_3\}$ do not contain r_3, the airs of these nodes remain the same. Next, the algorithm updates the airs of $\{r_4\}$, which results in $airs'(\{r_4\}) = \{\{r_2\}\}$. Finally, the algorithm updates the airs of $\{r_5\}$, which results in $airs'(\{r_5\}) = \{\{r_2, r_4\}\}$. Therefore, the goal is reachable. □

Deleting a can_revoke rule Suppose that *can_revoke*(T) is deleted from the policy. Since graph G_b is constructed using only *can_assign* rules, G_b remains the same and we simply update the airs of nodes as follows. First, starting from the first initial node V_0, the algorithm searches G_b along edges marked 1, for nodes whose airs may change. The airs of a node V may change if: (1) T is in the initial state and V does not contain T; or (2) there is an edge $V' \overset{\alpha}{\to} V$ such that $T \in V'$ and $T \notin V$. If such a node does not exist, the algorithm returns the previous analysis result. Otherwise, the algorithm updates the airs of the node as well as the airs of all nodes reachable from this node by edges marked 1 as follows: for every edge $e = V_1 \overset{\alpha}{\to} V_2$, if $airs'(V_1) = airs(V_1)$, then $airs'(V_2) = airs(V_2)$; otherwise, $airs'(V_2)$ is computed by removing sets in $airs(V_2)$ that are computed along e and recomputing airs along e using the non-incremental algorithm. If an edge marked 0 is encountered, the algorithm computes the set of airs along this edge. If the goal is not reachable from V_0, the algorithm picks another initial node and repeats the above process.

Example 10. *Consider the analysis problem instance in Example 2.*

Figure 2 gives the graph constructed using the non-incremental algorithm, which has two initial nodes: $\{r_4\}$ and $\emptyset$. $airs(\{r_4\}) = airs(\{r_6\}) = airs(\{r_5\}) = \{\emptyset\}$, and the airs of other nodes are $\emptyset$. Because $airs(\{r_5\})$ is not empty, the goal is reachable.

Suppose that can_revoke(r_4) is deleted from the policy. The incremental algorithm updates the graph from the initial node $\{r_4\}$. Because r_4 is in the initial RBAC policy and $\{r_6\}$ does not contain r_4, r_4 is added to $airs(\{r_6\})$. The algorithm then updates $airs(\{r_5\})$, which results in $airs'(\{r_5\}) = \emptyset$; the goal is no longer reachable. As a result, the algorithm picks another initial node $\emptyset$ and computes the airs of nodes reachable from $\emptyset$ using the non-incremental algorithm, which results in $airs'(\{r_5\}) = \{\emptyset\}$. Thus, the goal is reachable. □

Adding a can_assign rule Suppose that *can_assign*($P \wedge \neg N, T$) is added to the policy. If T is not a positive role relevant to the goal, the algorithm returns the previous analysis result; otherwise, the algorithm incrementally updates graph G_b and the airs of nodes.

In the first stage, starting from nodes that contain T, the algorithm computes all reachable edges enabled by the new rule. For each new edge $V \overset{\alpha}{\to} V'$, if V is a node in G_b and $airs(V) \neq \emptyset$, V is added to a set *affectedNodes*. Also, all new initial nodes are added to a set *newInit*. The new edges are marked 0, indicating that they have not been processed in the second stage.

If the previous analysis result is true, the algorithm returns true. Otherwise, the

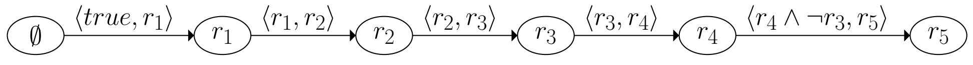

$$\begin{tikzcd}
\emptyset \arrow[r, "\langle true, r_1\rangle"] & r_1 \arrow[r, "\langle r_1, r_2\rangle"] & r_2 \arrow[r, "\langle r_2, r_3\rangle"] & r_3 \arrow[r, "\langle r_3, r_4\rangle"] & r_4 \arrow[r, "\langle r_4 \wedge \neg r_3, r_5\rangle"] & r_5
\end{tikzcd}$$

Fig. 12. The transition graph computed in Examples 9 and 11 using the non-incremental backward algorithm.

algorithm updates the airs of nodes as follows. For every node $V \in$ *affectedNodes*, it updates the airs of nodes reachable from V along new edges until a node containing T is encountered. The algorithm then updates the airs of this node as well as the airs of all nodes reachable from this node: for every edge $V_1 \overset{\alpha}{\to} V_2$, the algorithm computes $airs'(V_2)$ by adding sets that are computed from $airs'(V_1) \setminus airs(V_1)$ along the edge, to $airs(V_2)$. If $airs(goal) \neq \emptyset$, the algorithm returns true. Otherwise, the algorithm computes the airs of nodes reachable from the new initial nodes in *newInit*.

Example 11. *Consider the analysis problem in Example 1. The goal is not reachable. Figure 12 gives the graph constructed in the first stage.* $airs(\emptyset) = airs(\{r_1\}) = airs(\{r_2\}) = \{\emptyset\}$, $airs(\{r_3\}) = \{\{r_2\}\}$, $airs(\{r_4\}) = \{\{r_2, r_3\}\}$, *and* $airs(\{r_5\}) = \emptyset$.

Assume that rule $can\_assign(r_1, r_4)$ *is added to the above policy. In the first stage, the incremental algorithm adds a new edge* $\{r_1\} \overset{\langle r_1, r_4\rangle}{\to} \{r_4\}$ *to the graph and adds* r_1 *to affectedNodes. In the second stage, the algorithm starts from* $\{r_1\}$ *and updates the airs of nodes reachable from* $\{r_1\}$: $airs(\{r_4\}) = \{\{r_2, r_3\}, \{\emptyset\}\}$ *and* $airs(\{r_5\}) = \{\emptyset\}$. *Because* $airs(\{r_5\})$ *is not empty, the goal is reachable.* □

Deleting a can_assign rule Suppose that $can\_assign(P \wedge \neg N, T)$ is deleted from the policy. If T is not a positive role relevant to the goal, then deleting such a rule does not affect the analysis result; otherwise, the algorithm performs the following steps.

First, the algorithm back-chains from the goal and marks the following nodes as valid nodes: (1) the goal node, and (2) for every edge $V_1 \overset{\alpha}{\to} V_2$, if $\alpha \neq \langle P \wedge \neg N, T\rangle$ and V_2 is valid, then V_1 is valid. Valid nodes are nodes that remain in the graph after the rule is deleted. Next, for every edge $V_1 \overset{\langle P\wedge\neg N, T\rangle}{\to} V_2$, the algorithm deletes the sets in $airs(V_2)$ that are computed through the edge and adds V_2 to a set L_T. The algorithm then deletes all nodes not marked valid, edges containing at least one such node, and edges of the form $V_1 \overset{\langle P\wedge\neg N, T\rangle}{\to} V_2$. Finally, the algorithm updates the airs of nodes reachable from nodes in L_T: for every edge $V_1 \overset{\alpha}{\to} V_2$, $airs'(V_2)$ is computed by removing all sets from $airs(V_2)$ that are computed from $airs(V_1) \setminus airs'(V_1)$. If the goal was previously reachable but $airs'(goal) = \emptyset$, the algorithm computes airs of nodes that have not been processed using the non-incremental algorithm.

Note that, an alternative (and incorrect) approach to detecting invalidated nodes is to back-chain from the goal, delete edges computed through the deleted rule, delete nodes without outgoing edges, and then delete edges that contain deleted nodes. Such an approach will fail if the graph contains cycles: nodes in a cycle may not be reachable from the goal node after the rule is deleted, but still contain outgoing edges.

Example 12. *Consider the analysis problem in Example 2.*

The graph computed by the non-incremental algorithm is shown in Figure 2. The

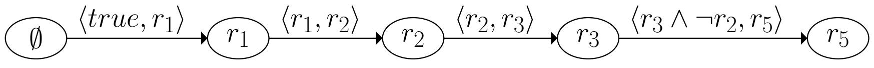

$\emptyset$
$\langle true, r_1\rangle$
r_1
$\langle r_1, r_2\rangle$
r_2
$\langle r_2, r_3\rangle$
r_3
$\langle r_3 \wedge \neg r_2, r_5\rangle$
r_5

Fig. 13. The graph computed by the incremental algorithm in Example 12.

graph has two initial nodes: $\{r_4\}$ and $\emptyset$. The algorithm first computes the airs of $\{r_4\}$ and nodes reachable from $\{r_4\}$: $airs(\{r_4\}) = airs(\{r_6\}) = airs(\{r_5\}) = \{\emptyset\}$. Because $airs(\{r_5\})$ is not empty, the goal is reachable. The airs of other nodes are empty.

Assume that $can\_assign(r_6 \wedge \neg r_4, r_5)$ is deleted. First, the incremental algorithm identifies valid nodes, which are $\{r_5\}$, $\{r_3\}$, $\{r_2\}$, $\{r_1\}$, and $\emptyset$. Next, the algorithm processes the edge $\{r_6\} \overset{\langle r_6\wedge\neg r_4, r_5\rangle}{\to} \{r_5\}$ and removes the sets in $airs(\{r_5\})$ that are computed along the edge, which results in $airs'(\{r_5\}) = \emptyset$. The algorithm then removes nodes that are not valid, i.e., $\{r_4\}$ and $\{r_6\}$, and removes edges $\{r_6\} \overset{\langle r_6\wedge\neg r_4, r_5\rangle}{\to} \{r_5\}$ and $\{r_4\} \overset{\langle r_4\wedge\neg r_3, r_6\rangle}{\to} \{r_6\}$. Because $airs'(\{r_5\})$ becomes empty, the algorithm picks another initial node $\emptyset$ and computes the airs of nodes reachable from $\emptyset$, which results in $airs'(\emptyset) = airs'(\{r_1\}) = airs'(\{r_2\}) = airs'(\{r_3\}) = airs'(\{r_5\}) = \{\emptyset\}$. Because $airs'(\{r_5\})$ is not empty, the algorithm returns true. The resulting graph is given in Figure 13. $\square$

3.6 Experimental Results

This section presents experimental results of our incremental analysis algorithms for ARBAC with the separate administration restriction. All reported data were obtained on a 2.5GHz Pentium machine with 4GB RAM running Linux 2.6.28.

3.6.1 Experimental Results: Incremental Forward Analysis Algorithms

We apply the non-incremental and our incremental forward algorithms to an ARBAC policy ψ_1, generated using a random policy generation algorithm with parameter values (e.g., the percentage of mixed roles) similar to the university ARBAC policy developed in [26]. The policy comprises 313 *can_assign* rules, 64 *can_revoke* rules, and 32 roles, among which 10 are administrative, 17 are positive, 8 are negative, and 8 are mixed.

We choose two goals $goal_1$ and $goal_2$ such that $goal_1$ is reachable from the initial state $\emptyset$ and $goal_2$ is unreachable, and the size of transition graphs constructed during analysis is reasonably large (155478 transitions for $goal_1$ and 225792 transitions for $goal_2$). We then randomly generate a set of operations that add rules to ψ_1 or delete rules from ψ_1, and use them to compare the performance of our incremental algorithms against the non-incremental algorithm.

Table 1 compares the execution time of the non-incremental algorithm without saving the graph (NonInc)[2], IncFwd1, IncFwd2, and LazyInc for goals $goal_1$ and $goal_2$, when a single change is made to policy ψ_1. Note that, with a single change, LazyInc works as IncFwd1 if the change does not affect the analysis result (except that the added rule is stored in *DelayedRule*); otherwise, it works as IncFwd2. Each

[2] The overhead for saving the transition graph is 0.11 seconds – 0.27 seconds.

Operation	States	Trans	Time(Sec.)			
			NonInc	IncFwd1	IncFwd2	LazyInc
add can_assign	30567	233299	81.68	0	5.96	0
delete can_assign	19983	153931	47.41	45.90	6.48	6.48
add can_revoke	20866	159290	49.43	0	0.72	0
delete can_revoke	20297	154853	47.96	11.87	0.56	0.56

(a)

Operation	States	Trans	Time(Sec.)			
			NonInc	IncFwd1	IncFwd2	LazyInc
add can_assign	33396	440052	164.01	161.79	100.91	100.91
delete can_assign	18000	220248	66.74	0	7.37	0
add can_revoke	19968	247834	76.16	21.23	7.66	7.66
delete can_revoke	18432	224928	68.62	0	0.50	0

(b)

Table 1. Performance results of NonInc, IncFwd1, IncFwd2, and LazyInc on ψ_1 for (a) $goal_1$ and (b) $goal_2$.

data point reported is an average over 32 randomly generated rules, except for the case "add *can_revoke*": because only 10 roles cannot be revoked in ψ_1, we generate only rules that revoke these 10 roles. Columns "States" and "Trans" give the average number of states and transitions computed using NonInc, respectively.

Results for adding/deleting a can_revoke rule Table 1 shows that, when a *can_revoke* rule is added, IncFwd2 is 68.27 and 9.94 times faster than NonInc for $goal_1$ and $goal_2$, respectively. When only the 2 rules that revoke mixed roles are considered, IncFwd2 is 15.12 times faster than NonInc for $goal_1$ and 2.78 times faster than NonInc for $goal_2$. When considering both goals, IncFwd2 is 14.27 and 2.41 times faster than NonInc and IncFwd1, respectively, and LazyInc is 15.62, 2.64, and 1.09 times faster than NonInc, IncFwd1, and IncFwd2, respectively.

When a *can_revoke* rule is deleted, IncFwd2 is 86.22 and 135.69 times faster than NonInc for $goal_1$ and $goal_2$, respectively. When considering only the 8 rules that revoke mixed roles, IncFwd2 is 21.39 times faster than NonInc for $goal_1$ and 33.8 times faster than NonInc for $goal_2$. When both goals are considered, IncFwd2 is 109.75 and 11.19 times faster than NonInc and IncFwd1, respectively, and LazyInc is 209.33, 21.33, and 1.91 times faster than NonInc, IncFwd1, and IncFwd2, respectively.

Results for adding/deleting a can_assign rule All *can_assign* rules added/deleted are relevant to the goal. Observe from Table 1 that, when a *can_assign* rule is added, IncFwd2 is 13.71 and 1.63 times faster than NonInc for $goal_1$ and

Goal	States	Trans	Time(Sec.)			
			NonInc	IncFwd1	IncFwd2	LazyInc
goal$_1$	30219	240802	84.40	33.09	3.72	2.97
goal$_2$	18613	228819	74.00	25.29	5.25	4.60

Table 2. Performance results of NonInc, IncFwd1, IncFwd2, and LazyInc for *goal*$_1$ and *goal*$_2$ on ten sequences of operations; in every sequence, only the last operation affects the analysis result.

Goal	States	Trans	Time(Sec.)			
			NonInc	IncFwd1	IncFwd2	LazyInc
goal$_1$	34134	268270	95.97	21.39	4.61	3.13
goal$_2$	21166	263731	83.76	19.96	4.22	3.18

Table 3. Performance results of NonInc, IncFwd1, IncFwd2, and LazyInc for *goal*$_1$ and *goal*$_2$ on ten sequences of operations, where each sequence contains 10 operations and any operation(s) in the sequence may affect the analysis result.

goal$_2$, respectively. This is because, the size of the transition graph increases less significantly for *goal*$_1$ than *goal*$_2$ (160550 vs 440052 transitions). In particular, for one of the 32 rules generated for *goal*$_2$, the size of the graph constructed after the rule is added is 10 times the size of the graph constructed before the rule is added. As a result, IncFwd2 computes a large number of new states and transitions using the non-incremental algorithm, and hence is only slightly faster than NonInc for this rule (1165 seconds vs 1199 seconds). When considering both goals, IncFwd2 is 2.32 and 1.53 times faster than NonInc and IncFwd1, respectively, and LazyInc is 2.46, 1.60, and 1.06 times faster than NonInc, IncFwd1, and IncFwd2, respectively.

When a *can_assign* rule is deleted, IncFwd2 is 7.31 and 9.04 times faster than NonInc for goals *goal*$_1$ and *goal*$_2$, respectively. When both goals are considered, IncFwd2 is 8.24 and 3.31 times faster than NonInc and IncFwd1, respectively, and LazyInc is 17.61, 7.08, and 2.14 times faster than NonInc, IncFwd1, and IncFwd2, respectively.

Results for adding/deleting a sequence of rules Table 2 compares the performance of the non-incremental algorithm and three incremental algorithms on 10 sequences of operations. In every sequence, only the last operation affects the analysis result. The column "Time" gives the average execution time of the algorithms for each operation. The results show that, when analyzing policy ψ_1 with *goal*$_1$, for each operation, LazyInc is 28.43, 11.14, and 1.25 times faster than NonInc, IncFwd1, and IncFwd2, respectively. When analyzing ψ_1 with *goal*$_2$, for each operation, LazyInc is 16.08, 5.50, and 1.14 times faster than NonInc, IncFwd1, and IncFwd2, respectively.

Table 3 compares performance of the non-incremental algorithm and three incremental algorithms on 10 sequences of operations where each sequence contains 10 operations and any operation(s) in the sequence may affect the analysis result.

The results show that, when analyzing policy ψ_1 with $goal_1$, for each operation, LazyInc is 30.69, 6.84, and 1.35 times faster than NonInc, IncFwd1, and IncFwd2, respectively. When analyzing ψ_1 with $goal_2$, for each operation, LazyInc is 26.32, 6.27, and 1.45 times faster than NonInc, IncFwd1, and IncFwd2, respectively.

Disk space consumption IncFwd1 does not require additional disk space. For IncFwd2, we associate each state with a unique ID and store mappings between IDs and states. For each transition, we store the IDs of the source state and the destination states, which ensures that each state is stored only once. The disk space for storing transition graphs for $goal_1$ and $goal_2$ are 4.83MB and 5.10MB, respectively.

3.6.2 Experimental Results: Incremental Backward Algorithm

Table 4 compares the execution time of non-incremental and incremental backward algorithms. The column heading "NonIncBack" refers to the non-incremental backward algorithm without storing the graph and the set of airs [3]. The column heading "IncBack" refers to the incremental backward algorithm. We choose a randomly generated policy ψ_2 (the parameter values are similar to those of ψ_1), two goals $goal_3$ and $goal_4$, and two initial states i_3 and i_4, so that (1) $goal_3$ is reachable from i_3 and $goal_4$ is not reachable from i_4; (2) the size of the graph is reasonably large (286009 and 87945 edges for $goal_3$ and $goal_4$, respectively); and (3) both stages of the algorithm are performed during analysis. The disk space for storing the graph is 5.33MB and 1.72MB for $goal_3$ and $goal_4$, respectively. The disk space for storing the airs is 1.67MB and 3.71MB for $goal_3$ and $goal_4$, respectively.

When a *can_revoke* rule is added or deleted, the graph remains the same. Table 4 shows that, when a *can_revoke* rule is added, IncBack is 7.97 and 1.56 times faster than NonIncBack for $goal_3$ and $goal_4$, respectively. When a *can_revoke* rule is deleted, IncBack is 8.10 and 18.04 times faster than NonIncBack for $goal_3$ and $goal_4$, respectively.

When a *can_assign* rule is added to the policy, IncBack is 12 times and 32.66 times faster than NonIncBack for $goal_3$ and $goal_4$, respectively. Saving the graph and airs imposes 0.51 seconds and 0.97 seconds of additional overhead for $goal_3$ and $goal_4$, respectively. When a *can_assign* rule is deleted, IncBack is 13.19 and 46.87 times faster than NonIncBack for $goal_3$ and $goal_4$, respectively.

4 Incremental User-role Reachability Analysis for ARBAC without Separate Administration

This section presents IncFwdWSA, an incremental forward algorithm for analyzing ARBAC policies without separate administration. Let $I = \langle UA_0, \psi, goal\rangle$ be the user-role reachability problem instance and $G(I)$ be the graph computed from I using the non-incremental algorithm. The algorithm is given below.

Add a can_assign rule Suppose that $can\_assign(r_a, P \wedge \neg N, T)$ is added to the policy. Let $I' = \langle UA_0, \psi \cup \{can\_assign(r_a, P \wedge \neg N, T)\}, goal\rangle$. Differences between IncFwdWSA and IncFwd2 are given below.

[3] the additional overhead for storing the graph and the set of airs is 0.58 seconds – 1.04 seconds.

Operation	Nodes	Edges	Time (NonIncBack)	Time (IncBack)
add can_revoke	37112	286009	22.81	2.86
add can_assign	37112	287046	46.14	3.6
delete can_revoke	37112	286009	46.06	5.69
delete can_assign	37085	285226	32.56	2.47

(a)

Operation	Nodes	Edges	Time (NonIncBack)	Time (IncBack)
add can_revoke	15566	87945	44.49	28.55
add can_assign	15566	88145	51	1.56
delete can_revoke	15566	87945	94.17	5.22
delete can_assign	15560	87774	43.52	0.93

(b)

Table 4. Performance comparison of NonIncBack and IncBack on ψ_2 for (a) $goal_3$ and (b) $goal_4$.

- In IncFwdWSA, we need to process both administrative roles and regular roles during slicing, while in IncFwd2, we process only regular roles during slicing.
- *IncRelRule* (i.e., the set of rules that are useful for reaching the goal) in IncFwd-WSA is defined as follows.
  (1) $can\_revoke(r'_a, r) \in IncRelRule$ if r is a mixed role in I' or is a negative role in UA_0.
  (2) $can\_assign(r_a, P \wedge \neg N, T) \in IncRelRule$.
  (3) a *can_assign* rule $\rho \in IncRelRule$ if
    (a) $target(\rho) \in Rel_+(I')$ and there exists $\rho' \in IncRelRule$ such that $target(\rho') \in poscond(\rho)$ or $target(\rho') = admin(\rho)$ where $admin(\rho)$ is the administrative precondition of rule ρ, or
    (b) $target(\rho)$ is a positive role relevant to T, or
    (c) $target(\rho)$ is a positive role relevant to r'_a.
- Theorem 1 is extended to consider multiple users.

Add a can_revoke rule Suppose that $can\_revoke(r_a, T)$ is added to the policy. Let $I' = \langle UA_0, \psi \cup \{can\_revoke(r_a, T)\}, goal\rangle$. Differences between IncFwdWSA and IncFwd2 are given below.

- In IncFwdWSA, we perform incremental slicing from r_a in a way similar to adding *can_assign*.
- *IncRelRule* in IncFwdWSA is defined as follows.

(1) $can\_revoke(r_a, T) \in IncRelRule$.
(2) $can\_revoke(r'_a, r) \in IncRelRule$ if r is a mixed role in I' or is a negative role in UA_0.
(3) a *can_assign* rule $\rho \in IncRelRule$ if
   (a) $target(\rho) \in Rel_+(I')$ and $T \in negcond(\rho)$ where $negcond(\rho)$ is the set of negative preconditions of ρ, or
   (b) $target(\rho) \in Rel_+(I')$ and there exists $\rho' \in IncRelRule$ such that $target(\rho) \in poscond(\rho')$ or $target(\rho) = admin(\rho')$ or
   (c) $target(\rho)$ is a positive role relevant to r_a or
   (d) $target(\rho)$ is a positive role relevant to r'_a

- We update the graph in a way similar to adding *can_assign*.

Deleting a can_assign rule Suppose that $can\_assign(r_a, P \wedge \neg N, T)$ is deleted from the policy. We summarize the differences between IncFwdWSA and IncFwd2 below.

- Theorem 2 is extended to consider multiple users.
- We remove invalidated transitions for multiple users.

Delete a can_revoke rule Suppose that $can\_revoke(r_a, T)$ is deleted from the policy. Differences between IncFwdWSA and IncFwd2 are given below.

- The algorithm must check if transitions revoking T are derivable through other *can_revoke* rules.
- We remove invalidated transitions for multiple users.

4.1 Experimental Results

This section presents experimental results of our incremental analysis algorithms without the separate administration restriction. All reported data was obtained on a 2.5GHz Pentium machine with 4GB RAM running Linux 2.6.28.

We apply the non-incremental algorithm and our incremental forward algorithms to the university ARBAC policy developed in [26]. We choose two initial RBAC policies UA_1 and UA_2, and goals $goal_1$ and $goal_2$ such that $goal_1$ is reachable from UA_1, $goal_2$ is not reachable from UA_2, and the transition graph is reasonably large. We then randomly generate a set of operations that add rules to the policy or delete rules from the policy, and compare the performance of our incremental algorithms against the non-incremental algorithm.

Table 1 compares the execution time of the non-incremental algorithm (NonIncWSA) against the incremental algorithm (IncFwdWSA) for the above policies and goals. Each data point reported is an average over 32 randomly generated rules. Columns "States" and "Trans" give the average number of states and transitions computed using NonIncWSA, respectively.

Results for adding/deleting a can_revoke rule Table 5 shows that, when a *can_revoke* rule is added, IncFwdWSA is 18.01 and 5.34 times faster than NonIncWSA for $goal_1$ and $goal_2$, respectively. When only the 8 rules that revoke mixed roles are

Operation	States	Trans	Time (NonIncWSA)	Time (IncFwdWSA)
add can_revoke	31256	171552	267.26	14.84
add can_assign	27661	131365	314.51	30.26
delete can_revoke	24107	127956	100.50	0.41
delete can_assign	24401	129570	101.58	2.41

(a)

Operation	States	Trans	Time (NonIncWSA)	Time (IncFwdWSA)
add can_revoke	42961	279603	252.01	47.23
add can_assign	38140	247887	220.10	40.6
delete can_revoke	38751	251573	216.16	0.65
delete can_assign	39366	255879	219.88	4.82

Table 5. Performance comparison of NonIncWSA and IncFwdWSA for (a) $goal_1$ and (b) $goal_2$

considered, IncFwdWSA is 12.87 times faster than NonIncWSA for $goal_1$ and 1.22 times faster than NonIncWSA for $goal_2$. When considering both goals, IncFwdWSA is 8.37 times faster than NonIncWSA. Saving the graph imposes 0.64 and 1.41 seconds of additional overhead for goals $goal_1$ and $goal_2$, respectively.

When a *can_revoke* rule is deleted, IncFwdWSA is 244.74 and 331.28 times faster than NonIncWSA for $goal_1$ and $goal_2$, respectively. When considering only 1 rule that revokes mixed roles, IncFwdWSA is 5.12 times faster than NonIncWSA for $goal_1$ and 10.36 times faster than NonIncWSA for $goal_2$. When both goals are considered, IncFwdWSA is 297.85 times faster than NonIncWSA. Saving the graph imposes 0.46 and 1.27 seconds of additional overhead for $goal_1$ and $goal_2$, respectively.

Results for adding/deleting a can_assign rule Table 5 shows that, when a *can_assign* rule is added, IncFwdWSA is 10.39 and 5.42 times faster than NonIncWSA for $goal_1$ and $goal_2$, respectively. When considering only 13 rules that add roles relevant to $goal_1$, IncFwdWSA is 8.40 times faster than NonIncWSA. When considering only 13 rules that add roles relevant to $goal_2$, IncFwdWSA is 2.21 times faster than NonIncWSA. When considering both goals, IncFwd2 is 7.54 times faster than NonIncWSA. The additional overhead of saving the graph is 0.68 and 1.27 seconds for $goal_1$ and $goal_2$, respectively.

When a *can_assign* rule is deleted, IncFwdWSA is 42.16 and 45.62 times faster than NonIncWSA for goals $goal_1$ and $goal_2$, respectively. When considering only the 4 rules that add positive roles, IncFwdWSA is 5.27 times faster and 5.70 times faster for $goal_1$ and $goal_2$, respectively.

When both goals are considered, IncFwdWSA is 44.47 times faster than Non-

IncWSA. The additional overhead of saving the graph is 0.47 and 1.29 seconds for $goal_1$ and $goal_2$ respectively.

Disk space consumption The amount of disk space used by NonIncWSA to store the transition graph is 17.39MB for $goal_1$, and 52.4MB for $goal_2$. These numbers are greater than the amount of disk space used by NonInc (Section 3.6) because the size of states in the graph generated by NonIncWSA is larger.

5 Other Analysis Problems

This section presents incremental analysis algorithms for *role-role containment*, *user-role availability*, *weakest precondition*, *dead role*, and *permission-role reachability.*

5.1 Incremental User-role Availability

User-role availability [17, 26] asks if a user u is always a member of a role r. Similar to the non-incremental algorithm, our incremental algorithm reduces the problem to the incremental user-role reachability analysis problem by adding a rule $can\_assign(\neg r, r')$ to the policy and then applying incremental user-role reachability analysis algorithms to check if r' is reachable. If so, the algorithm returns false; otherwise, the algorithm returns true.

5.2 Incremental Role-role Containment

Role-role containment [18, 26] asks: "in every state reachable from a given initial state, is every member of role r_1 also a member of role r_2"? Similar to the non-incremental algorithm, our incremental algorithm reduces the problem to the incremental user-role reachability analysis by adding rule $can\_assign(r_1 \wedge \neg r_2, r)$ to the policy and then applying the incremental user-role reachability analysis algorithm to check if r is reachable from the initial state. If so, the algorithm returns false; otherwise the algorithm returns true.

5.3 Incremental Weakest Precondition

The weakest precondition query returns the minimal sets of initial roles necessary for assigning the target user to the roles in the goal [26]. The weakest preconditions can be computed by constructing the backward graph and computing the smallest initial nodes (i.e., nodes that are subsets of the initial RBAC policy), from which the goal is reachable. As an optimization, the algorithm processes initial nodes in the ascending order of size.

Let $I = \langle \psi, goal \rangle$ be a weakest precondition analysis problem instance, G_b be the backward graph constructed for I, and $WP(I)$ be the weakest precondition computed for I. To support incremental analysis, we mark initial nodes from which the goal is not reachable with 0 and mark initial nodes from which the goal is reachable with 1. The incremental algorithm is given below.

Adding can_revoke/can_assign Assume that $can\_revoke(T)$ is added to ψ. First, we start from edges whose negative preconditions contain role T and compute a set UA_T of nodes that contain role T and are backward-reachable from the edges. If such a node does not exist, the algorithm returns the previous analysis result. Otherwise, we compute a set S_L of initial nodes smaller than $WP(I)$ that are backward reachable from each node in UA_T. The set of smallest nodes in S_L, from which the goal is reachable, is the weakest precondition.

Adding *can_assign*$(P \wedge \neg N, T)$ is handled similarly except that S_L consists of initial nodes smaller than $|WP(I)|$ that are backward reachable from new edges in the graph.

Deleting can_revoke/can_assign Assume that rule *can_revoke*(T) is deleted from ψ. If the goal is still reachable from some node in $WP(I)$, then the algorithm returns all nodes in $WP(I)$ from which the goal is reachable. Otherwise, we start from edges whose negative preconditions contain T and check if there exists a backward reachable node U_T that contains T. If such a node does not exist, the algorithm returns the previous analysis result. Otherwise, the algorithm processes initial nodes marked with 1. If the goal is not reachable from such nodes, then the algorithm computes the weakest precondition from nodes that have not been processed in the previous analysis, i.e. nodes not marked with 1 or 0.

Assume that rule *can_assign*$(P \wedge \neg N, T)$ is deleted from ψ. If the goal is still reachable from some nodes in $WP(I)$, then the algorithm returns nodes in $WP(I)$ from which the goal is reachable. Otherwise, the algorithm incrementally updates the graph and processes the set of initial nodes marked with 1. If the goal is not reachable, then the algorithm computes the weakest precondition from nodes that have not been processed in the previous analysis.

5.4 Incremental Dead Role Analysis

Dead role analysis [8] computes a set of roles that cannot be assigned to any user, i.e. *dead roles*. Presence of dead roles may indicate errors in the policy specification such as missing rules. A straightforward approach to dead role analysis is to compute a set of roles that can be assigned to each user until all roles have been assigned or all users have been considered. Gofman et. al. [8] proposed the following optimizations: (1) apply the slicing transformation to eliminate roles and rules not useful for adding unassigned roles; (2) consider only users with distinct set of roles; and (3) first consider users who are assigned most roles that are positively relevant to unassigned roles; such users can potentially be assigned to most unassigned roles.

To enable incremental dead role analysis, we extend the non-incremental algorithm in [8] to store the set of dead roles. If there are no dead roles, we mark the last user considered and all users that have not been considered with 0. We then handle changes to the ARBAC policy as follows.

Adding can_revoke/can_assign If the policy does not contain dead roles, then adding an ARBAC rule does not affect the analysis result. Otherwise, we pick a user that is not marked with 0 and is assigned most roles that are positively relevant to the added rule, incrementally update the user's graph using LazyInc, and update the set of dead roles. If all users not marked with 0 have been processed and the set of dead roles is not empty, then we pick a user that is marked with 0 and is assigned most roles that are positively relevant to the added rule, and perform non-incremental analysis. Repeat the above process until the set of dead roles is empty or all users have been processed.

Deleting can_revoke/can_assign Assume that $can\_assign(P \wedge \neg N, T)$ is deleted from the policy. First, we compute a set of roles A that are not in the initial policy and are reachable through T. We then pick a user not marked with 0 whose graph contains most roles in A, incrementally update the graph using LazyInc, and remove roles that are assigned to users from A. Repeat the above process until A is empty or all users not marked with 0 have been processed. If A is empty, then the algorithm returns the previous analysis result. Otherwise, we process all users marked with 0 using the non-incremental algorithm.

Deleting rule $can\_revoke(T)$ is handled similarly except that A contains a set of roles that are enabled by postconditions of *can_assign* rules whose negative precondition is T.

5.5 Incremental Permission-role Reachability Analysis

The permission-role reachability analysis problem [24] asks, "Can administrators in administrative roles in A assign a permission p to all roles in goal?". Since RBAC and ARBAC specifications for the user-role and permission-role assignments are symmetrical, incremental permission-role reachability analysis can be performed in exactly the same manner as incremental user-role reachability analysis, except user-role relations are replaced with permission-role relations.

6 Related Work

A number of researchers investigated the problem of analyzing (a subset of) static ARBAC policies. Li et al. [18] presented algorithms and complexity results for analysis of two restricted versions of ARBAC97 – AATU and AAR. This work did not consider negative preconditions in ARBAC policies. Schaad and Moffett [25] used the Alloy analyzer [13] to check separation of duty properties for ARBAC97. However, they did not consider preconditions in ARBAC policies. Sasturkar et al. [24] and Jha et al. [15] presented algorithms and complexity results for analysis of ARBAC policies subject to a variety of restrictions. Stoller et. al. [26] proposed the first fixed-parameter-tractable algorithms for analyzing ARBAC policies, which lay the groundwork for incremental analysis algorithms presented in this paper. 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. Ferrara et al [5] proposed to convert ARBAC policies to imperative programs and apply abstract-interpretation techniques to perform security analysis. Later, they proved that if the goal is reachable in an ARBAC policy, then there exists a run with at most $|\text{administrative roles}| + 1$ users in which the goal is reachable [6]. Becker [2] proposed a language DYNPAL for specifying dynamic authorization policies, which is more expressive than ARBAC, and presented techniques for analyzing DYNPAL. Yang et al. [29] developed a number of reduction techniques and parallel analysis algorithms based on the algorithm in [26], which significantly improves the scalability of the algorithm. Stoller et al. [27] presented algorithms for analyzing parameterized ARBAC. Uzun et al. [28] developed algorithms for analyzing temporal role-based access control models. Our work in this paper is different from the above as we consider changes to ARBAC policies.

Crampton [4] showed the undecidability of reachability analysis for RBAC, whose changes are controlled by commands consisting of pre-conditions and administrative operations; some commands are not expressible in the form allowed in ARBAC97 and some commands use administrative operations that change the ARBAC policy.

Prior works [12, 22, 16] have also considered changes to access control policies. However, their changes are not controlled by ARBAC, nor did these works consider changes to administrative policies. Fisler *et al.* [7] developed a change-impact analysis algorithm for RBAC policies based on binary decision diagram (BDD) by computing the semantic difference of two policies and checking properties of the difference. We consider changes to ARBAC policies, instead of RBAC policies. Furthermore, we also propose a lazy analysis algorithm.

Incremental computation has been studied in several areas, including deductive databases, logic programming, and program analysis. However, to the best of our knowledge, we are the first to develop the incremental algorithms for ARBAC policy analysis. Gupta et al. [10] incrementally updated materialized views in databases by counting the number of derivations for each relation. Our approach is more efficient for analyzing ARBAC policies: we compute each transition only once; counting derivations would require determining all ways in which a transition can be computed. Gupta et al. [11] proposed a two-phase delete-rederive algorithm, which first deletes relations that depend on the deleted relation, and then rederives the deleted relations that have alternative derivations. Similar approaches were adapted in [21]. We avoid the rederivation phase by removing only those roles from the state for which all derivations have been invalidated. Lu et al. [19] proposed a Straight Delete algorithm (StDel) which eliminates the rederivation phase of delete-rederive algorithm. Direct application of StDel to ARBAC policy analysis would require storing all derivations for every state and every transition and, just as with counting, would be less efficient. Conway et. al. [3] developed algorithms to incrementally update control flow graphs of C programs. Since ARBAC has no control flow, their algorithms are not directly applicable to our problem. All aforementioned work, in contrast to our algorithms, computed the exact data structure. Further, none of them have proposed a lazy algorithm as we do.

This paper extends our conference paper [9] in several ways. (1) we add incremental algorithms for availability, permission-role reachability, role-role containment, weakest precondition, and dead role analysis; (2) we develop algorithms for user-role reachability analysis for evolving ARBAC without separate administration restriction, implement the algorithms, and present the performance results; (3) we add proofs for Theorems 1 and 2; (4) we significantly reduce the disk space used for storing the graph (by up to 8.43x) by storing each state only once; our implementation in [9] explicitly stores the states in each transition, which results in a single state being stored multiple times; (5) we have improved the execution time of both forward and backward algorithms by using an indexing data structure, which is more efficient then using a full-fledged hashtable as we did in [9]; (6) we add pseudocodes for the forward algorithms for deleting *can_revoke* and *can_assign* rules and add examples for some of the algorithms; and (7) we have re-implemented the incremental backward algorithms.

7 Conclusion

This paper presents a number of incremental algorithms for user-role reachability analysis of evolving ARBAC policies. Our incremental algorithms identify changes to the policy that may affect the analysis result and use the information computed in the previous analysis to update the result. The experimental data shows that our incremental algorithms outperform the non-incremental algorithm in terms of execution time. We have also presented incremental algorithms for other analysis problems, including user-role availability, permission-role reachability, role-role containment, weakest precondition, and dead role analysis. Although our incremental algorithms are developed based on the analysis algorithm in [26], the general idea can be adapted to other non-incremental analysis algorithms as well.

In the future, we plan to further improve the performance of our incremental analysis algorithms. A promising optimization is not to perform operations, which do not affect the analysis result, on the graph. Such operations include operations that remove irrelevant roles from the graph and operations that change visible transitions to invisible transitions.

Acknowledgment

This work was supported in part by United States National Science Foundation Grant CNS-0855204. The authors thank Ruiqi Luo, Jian He, Jian Zhang, Ramyashree Rangaiyengar, and Arunpriya Somasundaram for their contributions to the implementation.

References

[1] F. Alberti, A. Armando, and S. Ranise. Efficient symbolic automated analysis of administrative attribute-based rbac-policies. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 165–175, New York, NY, USA, 2011. ACM.

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

[3] C. Conway, K. Namjoshi, D. Dams, and S. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *Computer Aided Verification*, 2005.

[4] J. Crampton. Authorizations and antichains, ph.d. thesis, university of london. 2002.

[5] A. L. Ferrara, P. Madhusudan, and G. Parlato. Security analysis of role-based access control through program verification. In *IEEE 25th Computer Security Foundations Symposium*, pages 113–125, 2012.

[6] 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.

[7] 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.

[8] M. I. Gofman, R. Luo, A. C. Solomon, Y. Zhang, P. Yang, and S. D. Stoller. Rbac-pat: A policy analysis tool for role based access control. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 46–49, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] M. I. Gofman, R. Luo, and P. Yang. User-role reachability analysis of evolving administrative role based access control. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 455–471, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Workshop on Deductive Databases*, pages 185–194, 1992.

[11] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *International Conference on Management of Data*, pages 157–166, 1993.

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

[13] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. pages 730–733, June 2000.

[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, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough. Towards formal verification of role-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 5(2), 2008.

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

[17] N. Li, J. C. Mitchell, and W. H. Winsborough. Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM*, 52(3):474–514, 2005.

[18] 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.

[19] J. Lu, G. Moerkotte, J. Schu, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views, 1995.

[20] D. F. F. R. Sandhu and D. R. Kuhn. The NIST model for role based access control: Towards a unified standard. In *ACM SACMAT*, pages 47–63, 2000.

[21] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *In International Conference on Logic Programming*, pages 392–406, 2003.

[22] R. Sandhu. The typed access matrix model. In *Proc. IEEE Symposium on Security and Privacy*, pages 122–136, 1992.

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

[24] A. Sasturkar, P. Yang, S. D. Stoller, and C. Ramakrishnan. Policy analysis for administrative role based access control. In *19th IEEE Computer Security Foundations Workshop*, 2006.

[25] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Proc.of SACMAT*, pages 13–22, 2002.

[26] S. Stoller, P. Yang, C. R. Ramakrishnan, and M. Gofman. Efficient policy analysis for administrative role based access control. In *ACM CCS*, pages 445–455, 2007.

[27] 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.

[28] 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.

[29] P. Yang, M. Gofman, and Z. Yang. Policy analysis for administrative role based access control without separate administration. In *the 27th IFIP WG 11.3 Con-*

ference on Data and Applications Security and Privacy (DBSEC13), pages 49 – 64, 2013.