EFFICIENT POLICY ANALYSIS FOR ADMINISTRATIVE ROLE BASED ACCESS CONTROL

BY

MIKHAIL I. GOFMAN

MS, Binghamton University, 2008
BS, Binghamton University, 2006

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate School of Binghamton University
State University of New York
2012
Accepted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2012

April 26, 2012

Ping Yang, Chair and Faculty Advisor
Department of Computer Science, Binghamton University

Leslie Lander, Member
Department of Computer Science, Binghamton University

Kyoung-Don Kang, Member
Department of Computer Science, Binghamton University

Scott Craver, Outside Examiner
Department of Electrical and Computer Engineering, Binghamton University, Binghamton University

iii
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. It is often difficult to fully understand the effect of an ARBAC policy by simple inspection, because sequences of changes by different administrators may interact in unexpected ways. ARBAC policy analysis algorithms can help by answering questions, such as user-role reachability, which asks whether a given user can be assigned to given roles by given administrators. This problem is intractable in general. The object of this dissertation is to develop efficient algorithms for analysis of ARBAC policies.

First, we present user-role reachability analysis algorithms for ARBAC that are fixed parameter tractable. These algorithms have a high complexity with respect to some parameter $k$ that is often small in practice. We have measured the performance of our analysis algorithms on a university policy, a healthcare policy, and families of synthetic policies in order to validate our parameterized complexity results. In addition, we have considered other analysis problems, including role containment and weakest preconditions.

Next, we present the first known incremental algorithms for analysis of evolving ARBAC policies. 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. Our incremental algorithms determine if a change may affect the analysis result, and if so, reuse the information of the previous analysis to incrementally update the analysis result. Detailed evaluations show that our incremental algorithms outperform the non-incremental algorithm in terms
of execution time.

Finally, we present incremental algorithms for analyzing information flows resulting from the RBAC policy. Information flow analysis can detect security breaches where information flows from a high security object to a lower security object. All of our incremental algorithms have the same or better worst-case complexity than our non-incremental algorithm and in practice significantly outperform the non-incremental algorithm.
Contents

List of Tables ................................................................. viii
List of Figures ............................................................... ix

1 Introduction ............................................................... 1

2 Preliminaries ............................................................ 4
  2.1 Role Based Access Control (RBAC) ......................... 4
  2.2 Administrative Role Based Access Control (ARBAC) ... 5

3 Efficient Policy Analysis for Administrative Role Based Access Control .... 8
  3.1 User-Role Reachability Under Separate Administration ...... 8
    3.1.1 Parameterized Complexity .................................. 9
    3.1.2 Fixed-Parameter Tractability of Reachability Under Separate Administration ................. 10
    3.1.3 Fixed-Parameter Tractability of Reachability With One Positive Precondition Under Separate Administration ......................... 16
  3.2 Beyond the Separate Administration Restriction ............... 23
    3.2.1 Fixed-Parameter Tractability of Reachability .............. 23
    3.2.2 Hierarchical Role Assignment ................................ 25
  3.3 Other Analysis Problems .......................................... 26
  3.4 Case Studies ........................................................ 27
  3.5 Experimental Results ............................................ 30

4 User-Role Reachability Analysis of Evolving Administrative Role Based Access Control .... 33
  4.1 User-Role Reachability Under Separate Administration .......... 33
    4.1.1 Incremental Forward Algorithms ......................... 33
    4.1.2 Incremental Algorithm: IncFwd1 ......................... 33
    4.1.3 Incremental Algorithm: IncFwd2 ......................... 34
    4.1.4 Lazy Incremental Forward Algorithm ..................... 47
    4.1.5 Incremental Backward Algorithm ......................... 49
    4.1.6 Experimental Results Under Separate Administration .... 54
    4.1.7 Experimental Results: Incremental Forward Analysis Algorithms ................... 54
    4.1.8 Experimental Results: Incremental Backward Algorithm ...... 56
  4.2 Incremental User-role Reachability Analysis for ARBAC without Separate Administration .... 57
    4.2.1 Experimental Results ...................................... 60
  4.3 Other Analysis Problems .......................................... 62
    4.3.1 User-role Availability ...................................... 62
    4.3.2 Role-role Containment ...................................... 62
4.3.3 Weakest Precondition ........................................ 62
4.3.4 Dead Role Analysis ........................................ 64

5 Incremental Information Flow Analysis of Role Based Access Control 66
  5.1 Information Flow Analysis of RBAC .......................... 67
  5.2 Incremental Information Flow Analysis ....................... 71
  5.3 Extensions .................................................. 76
    5.3.1 Information Flow Analysis of Parameterized Role Based Access Control .......................... 76
    5.3.2 Supporting Queries ...................................... 77
    5.3.3 Adding/Deleting a Sequence of Relations .................. 78
  5.4 Performance Results ...................................... 79

6 RBAC-PAT: A Policy Analysis Tool for Role Based Access Control 81
  6.1 Hierarchy Converter ........................................ 81
  6.2 Policy Analysis Engine ..................................... 81
  6.3 Case Studies ............................................ 82

7 Related Work .................................................. 84

8 Research Plan .................................................. 89

9 Milestones ..................................................... 91

Bibliography ..................................................... 91
List of Tables

3.1 (a) Running time of forward algorithm vs. number of mixed roles with $|R| = 32$. (b) Running time of forward algorithm vs. number of roles with 5 mixed roles. ................................................. 31

3.2 Performance of backward algorithm for (a) varying goal size and (b) varying number of roles with $|\text{Irrev}|/|R| = 0.05$ and $|\text{goal}| = 1$. .............. 32

4.1 Performance results of NonInc, IncFwd1, and IncFwd2 on $\psi_1$ for goals $\text{goal}_1$ and $\text{goal}_2$. ................................................................. 55

4.2 Performance results of NonInc, IncFwd1, IncFwd2, and LazyInc for $\text{goal}_1$ and $\text{goal}_2$ on ten sequences of operations. ................. 56

4.3 Performance comparison of NonIncBack and IncBack on $\psi_2$ for $\text{goal}_3$ and $\text{goal}_4$. ................................................................. 57

4.4 Performance comparison of NonIncWSA and Inc for $\text{goal}_1$ and $\text{goal}_2$ . . 60

5.1 Performance comparison of the non-incremental algorithm and the incremental algorithms .................................................. 80
List of Figures

3.1 (a) Part of state space generated without reduction and without slicing (32 states, 96 transitions). (b) State space generated without reduction and with slicing (8 states, 16 transitions). (c) State space generated with reduction and without slicing (3 states, 3 transitions). ua and ur abbreviate UserAssign and UserRevoke, respectively. The state space generated with reduction and slicing (not shown) contains 1 state, namely, \{r_1, r_2, r_3, r_4\}, and 0 transitions. 

3.2 The reduced transition graph constructed in Example 1 by the forward algorithm.

3.3 The graph constructed in Example 3 by the backward algorithm.

3.4 Graph constructed in Example 4.

4.1 Pseudocode for adding can_revoke(T).

4.2 The transition graph computed in Example 5 by the incremental algorithm.

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

4.4 Pseudocode for deleting can_revoke(T).

4.5 Pseudocode for adding can_assign(P ∧ ¬N, T).

4.6 The transition graph computed in Example 7 by the incremental algorithm.

4.7 Pseudocode for deleting can_assign(P ∧ ¬N, T).

4.8 The transition graph computed in Example 8 by the incremental algorithm.

4.9 Graphs computed in Examples 10 and 12: by the non-incremental backward algorithm.

4.10 The graph computed by the incremental algorithm in Example 13.

5.1 (a) The information flow graph generated using the algorithm in [33]. (b) The information flow graph generated using Algorithm 1. (c) The information flow graph containing only objects.

5.2 Information flow graph generated from the PRBAC policy in Example 16.

5.3 Information flow query for the university RBAC policy.

6.1 System architecture.

9.1 Gantt chart of the project milestones.
1 Introduction

Role Based Access Control (RBAC) has been widely used for limiting access to system resources to only authorized users [38]. RBAC specifies access controls by assigning users to roles and then associating each role with a set of permissions. A role represents a job function or a position within the organization. A user has a permission only if he is assigned to some role with that permission. In large organizations RBAC policies are usually manged by multiple administrators. Administrative Role Based Access Control (ARBAC) is used to specify the permissions of administrators to change RBAC policy.

Correct understating of RBAC and ARBAC policies is critical for assuring the security of the system. In practice, however, these policies are often too large and complex to be comprehended through manual inspection alone. This is because the changes made by multiple administrators may interact in unexpected ways. Automated policy analysis helps administrators understand the policy and detect potential flaws by answering queries about the policy such as user-role reachability analysis which asks: ”given an RBAC policy and an ARBAC policy, a set of administrators $A$, a target user $u$, and a set of roles (called the “goal”), is it possible for administrators in $A$ to assign the target user $u$ to roles in the goal?” It has been shown that the reachability analysis for ARBAC (with fixed role hierarchy) is PSPACE-complete [39; 23].

The main contributions of this dissertation can be summarized as follows:

1. We present an efficient forward algorithm for user-role reachability analysis of general miniARBAC policies (see Section 3.1.2) [46], and show that it is fixed-parameter tractable with respect to the number of mixed roles (roles that appear both negatively and positively in the policy). The algorithm uses forward search optimized using a reduction. We describe how to slice a policy with respect to a goal, to help avoid
exploration of irrelevant states.

2. We give an efficient backward-search algorithm for user-role reachability analysis for miniARBAC policies with at most one positive precondition per rule [46]. The algorithm is fixed-parameter tractable with respect to the number of irrevocable roles and has a similar tractability property with respect to the size of the goal (see Section 3.1.3). We also present a partial-order reduction to optimize the backward search.

3. The above algorithms rely on an aspect of ARBAC97 that we call separate administration, which requires that administrative roles and regular roles are disjoint. Prior work on ARBAC policy analysis [41; 28; 39] generally makes this assumption, but it is unrealistic in many cases. We lift this restriction and explore two approaches to policy analysis in this more general setting, by identifying conditions under which the general problem can be reduced to policy analysis with separate administration, and by extending our forward analysis algorithm to handle the general problem (see Section 3.2).

4. We describe two case studies: ARBAC policies for a university and a health-care institution (see Section 3.4). We observe several structural properties of them, and relate them to the assumptions and complexity parameters of our algorithms [46].

5. We measure the performance of our analysis algorithms on families of synthetic (randomly generated) policies, in order to validate our parameterized complexity results, determine whether the worst-case complexity manifests itself, and compare the performance of the forward and backward analysis algorithms when both apply (see Section 3.5).

6. We also consider other analysis problems, including role containment (is every member of role $r_1$ also a member of a role $r_2$ in all reachable policy states?) [28] and weakest preconditions (what are minimal sets of initial roles for a user, in order for that user to get added to roles in the goal?) (see Section 4.3) [46].
7. We present incremental algorithms for analysis of evolving ARBAC policies. Our incremental algorithms reuse 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 reused to update the result more quickly than a complete reanalysis.

8. We optimize the algorithm for analyzing the flows of information resulting from the RBAC policy [33] and then use the resulting algorithm to develop incremental algorithms.

9. We develop a Role Based Access Control Policy Analysis Tool (RBAC-PAT) [12] for analyzing various properties of RBAC and ARBAC policies including user-role reachability, user-role availability, role-role containment, weakest precondition, dead roles, and information flow.

10. For the final dissertation we plan to optimize analysis of ARBAC policies without separate administration (see Chapter 8).

The rest of this proposal is organized as follows. Chapter 2 gives the preliminaries. Chapter 3 presents algorithms for efficient analysis of ARBAC policies. These algorithms are then used in Chapter 4 to develop efficient algorithms for analysis of evolving ARBAC policies. Chapter 5 then presents efficient algorithms for analyzing information flows resulting from the RBAC policy. Chapter 6 presents RBAC-PAT: a graphical policy analysis tool based on the algorithms in Chapters 3 and 5. Finally, Chapter 7 gives the related work and the future research plans appear in Chapter 7.
2 Preliminaries

2.1 Role Based Access Control (RBAC)

The central notion of RBAC is that users are assigned to appropriate roles, and roles are assigned appropriate permissions. In this proposal, we study policy analysis for models of RBAC based on [1]. Following Sasturkar et al. [39], we adopt a simplified model, called miniRBAC, that does not support sessions, because the policy analysis queries we consider are independent of sessions.

miniRBAC. A miniRBAC policy $\gamma$ is a tuple $\langle U, R, P, UA, PA \rangle$ where

- $U, R$ and $P$ are finite sets of users, roles, and permissions, respectively. A permission represents authorization to invoke a particular operation on a particular resource.
- $UA \subseteq U \times R$ is the user-role assignment relation. $\langle u, r \rangle \in UA$ means that user $u$ is a member of role $r$.
- $PA \subseteq P \times R$ is the permission-role assignment relation. $\langle p, r \rangle \in PA$ means that members of role $r$ are granted the permission $p$.

The miniHRBAC model based on Hierarchical RBAC [1] extends the miniRBAC model with role hierarchies, which are a natural means for structuring roles to reflect an organization’s lines of authority and responsibility.

miniHRBAC. A miniHRBAC policy $\gamma_h$ is a tuple $\langle U, R, P, UA, PA, \succeq \rangle$ where $U, R, P, UA$ and $PA$ are as in miniRBAC, and $\succeq \subseteq R \times R$ is a partial order on the set $R$ of roles.

$r_1 \succeq r_2$ means $r_1$ is senior to $r_2$, i.e., every member of $r_1$ is also a member of $r_2$, and every permission assigned to $r_2$ is also available to members of $r_1$. Thus, $r_2$ inherits all the users of $r_1$ and $r_1$ inherits all the permissions of $r_2$. A user $u$ is an explicit member of a role $r$ if $\langle u, r \rangle \in UA$. A user $u$ is an implicit member of a role $r$ if $\langle u, r' \rangle \in UA$ for some $r'$.
such that $r' \succeq r$ and $r' \neq r$.

### 2.2 Administrative Role Based Access Control (ARBAC)

ARBAC97 is a classic model for decentralized administration of RBAC policies [37]. ARBAC97 has three components: (1) user-role administration (URA), (2) permission-role administration (PRA), and (3) role-role administration (RRA) for administration of the role hierarchy. We consider a modified version of ARBAC97 called miniARBAC similar to [39]. miniARBAC includes only the URA component; the permission assignment and role hierarchy are considered fixed. Extending our analysis techniques to handle changes to them is discussed at the end of this section. Since we do not consider changes to the role hierarchy, we adopt a simple ARBAC model without authority ranges [37], scopes [25; 7], or administrative domains [26]; policies using those features can be expressed without them when the role hierarchy is fixed.

The URA policy controls changes to the user-role assignment $UA$. Permission to assign users to roles is specified by the relation $can\_assign \subseteq R \times C \times R$, where $C$ is the set of all preconditions (called prerequisite conditions in [37]) over $R$. The precondition is a conjunction of literals, where each literal is either $r$ or $\neg r$ for some role $r$ in $R$. Given a miniRBAC policy $\gamma$ and a user $u$, $u$ satisfies a precondition $\wedge_i l_i$, denoted $u \models \gamma \wedge_i l_i$, iff for all $i$, either $l_i$ is a role $r$ and $u$ is a member of $r$ in $\gamma$, or $l_i$ is a negated role $\neg r$ and $u$ is not a member of $r$ in $\gamma$. A $UserAssign(r_a, u, r)$ action specifies that an administrator who is a member of the administrative role $r_a$ adds user $u$ to role $r$. This action is enabled in state $\gamma = \langle U, P, R, UA, PA \rangle$ iff there exists a precondition $c$ such that $\langle r_a, c, r \rangle \in can\_assign$ and $u \models \gamma c$. Execution of this action transforms $\gamma$ to the state $\gamma' = \langle U, P, R, UA \cup \{ \langle u, r \rangle \}, PA \rangle$.

Permission to revoke users from roles is specified by the relation $can\_revoke \subseteq R \times R$. This is analogous to $can\_assign$, except that it does not allow preconditions, because there is little evidence that preconditions for revocation are useful [37]. The action $UserRevoke(r_a, u, r)$ is defined similarly to $UserAssign$. 

5
**miniARBAC policy.** A miniARBAC policy is represented as $\psi = (\text{can Assign}, \text{can Revoke})$. We often refer to tuples in these two relations as *rules*. The role $r$ to which users or permissions are being assigned or removed (*i.e.*, the third component of the tuple) is called the *target* of the rule. A miniARBAC policy specifies a transition relation between miniRBAC (or miniHRBAC) policies, which we often refer to as *states*. A transition is denoted by $\gamma \xrightarrow{a} \psi \gamma'$, where $a$ is one of the four administrative actions defined above. When we do not care about the action $a$, we omit it from the transition, and when the miniARBAC policy $\psi$ is clear from context, we omit it from the transition.

**Separate Administration.** A role $r$ is an *administrative role* if it has an administrative permission, *i.e.*, there is a $\text{can Assign}$ tuple with $r$ in the first component. A role $r$ is a *regular role* if it has a regular permission, *i.e.*, there is a $\text{PA}$ tuple with $r$ in the first component. Our framework, like SARBAC [7], UARBAC [26], and Oracle, allows a role to be both a regular role and an administrative role. This flexibility allows many policies to be expressed more naturally. For example, in a university, a department chair has both regular permissions (*e.g.*, authorize expenses from the department’s accounts) and administrative permissions (*e.g.*, appoint faculty to committees).

Earlier work on ARBAC, such as ARBAC97 and ARBAC02 [37; 32], requires that (1) regular roles and administrative roles are separate (*i.e.*, no role is in both categories), and (2) in every $\text{can Assign}$ tuple, the first component is an administrative role, the condition mentions only regular roles, and the target is a regular role. We call this the *separate administration* restriction. Work on ARBAC policy analysis [41; 39; 28] generally adopts this restriction, with the exception of analysis for the AATU model in [28].\(^1\) This helps simplify a difficult problem, enabling steps towards more general solutions. The AATU model in [28] does not adopt this restriction, but adopts two other big restrictions instead (no revocation and no negative preconditions).

The analysis algorithms in Sections 3.1.2 and 3.1.3 take advantage of the separate ad-

---

\(^1\)The restriction adopted in the AAR model in [28] is similar although not identical.
ministration restriction. Section 3.2 tackles policy analysis without this restriction.
3 Efficient Policy Analysis for Administrative Role Based Access Control

3.1 User-Role Reachability Under Separate Administration

The user-role reachability problem is: Given an initial miniRBAC policy \( \gamma_0 = (R, UA_0) \), a miniARBAC policy \( \psi \), a set \( U_0 \) of users, a user \( u_t \) in \( U_0 \) (called the “target user”), and a set \( \text{goal} \) of roles, can the users in \( U_0 \) together transform \( \gamma_0 \) (under the restrictions imposed by \( \psi \)) to another miniRBAC policy \( \gamma \) in which \( u_t \) is a member of all roles in \( \text{goal} \)? A sequence of administrative actions (i.e., UserAssign and UserRevoke actions) that performs such a transformation is called a plan for (or solution to) the problem instance. We will often refer to user-role reachability simply as “reachability”. A reachability problem instance can be represented as a tuple \( \langle \gamma_0, \psi, U_0, u_t, \text{goal} \rangle \). We sometimes refer to miniRBAC policies as “states”.

Reachability Under Separate Administration. The separate administration restriction allows the specification of a problem instance to be simplified. When adopting this restriction, we also assume, without loss of generality, that the target user \( u_t \) is initially a member of regular roles only (if the target user were initially a member of administrative roles, we could give those roles to another user in \( U_0 \) instead, without affecting the answer to the reachability problem). Note that the separate administration restriction implies that the target user cannot be added later to an administrative role.

We call the other users in \( U_0 \) “administrators” (we could require that they are members of administrative roles only, but allowing them to be members of regular roles as well has no impact on the analysis). With this restriction, it is sufficient to consider only administrative actions by the administrators that change the target user’s role memberships. Let \( A \) be the set of administrative roles of users in \( U_0 \) in the initial state \( \gamma_0 \). We can merge those roles
into a single administrative role with the union of the administrative permissions of roles in \( A \), and eliminate all other administrative roles. We can then make this single administrative role implicit, \textit{i.e.}, we can eliminate the first component of the \textit{can assign} and \textit{can revoke} relations. Similarly, because role memberships of different users are independent, we can eliminate all users other than the target user. We can then make this user implicit, \textit{i.e.}, we can eliminate the first component of the user assignment \( UA \). We can also eliminate the first two parameters of administrative actions.

With these simplifications, a reachability problem instance can be represented as a tuple \( \langle \gamma_0, \psi, \text{goal} \rangle \) where \( \gamma_0 = \langle R, UA_0 \rangle \) is a simplified miniRBAC policy, \( \psi = \langle \text{can assign}, \text{can revoke} \rangle \) is a simplified miniARBAC policy, and \( \text{goal} \subseteq R \). Since the set of roles is fixed, we sometimes elide it, representing a state as \( UA \), instead of \( \langle R, UA \rangle \).

### 3.1.1 Parameterized Complexity

Parameterized complexity \cite{8} is an approach to deal with computationally difficult problems. The idea is to identify an aspect of the input that makes the problem computationally difficult, introduce a parameter to measure that aspect of the input, and develop a solution algorithm that may have high complexity in terms of that parameter, but has polynomial complexity in terms of the overall input size when the value of that parameter is fixed. This is called fixed-parameter tractability. Formally, a problem is \textit{fixed-parameter tractable (FPT)} with respect to parameter \( k \) if there exists an algorithm that solves it in \( O(f(k) \times n^c) \) time, where \( f \) is an arbitrary function (depending only on its argument \( k \)), \( n \) is the input size, and \( c \) is a constant.

We say that a problem is \textit{fixed-parameter polynomial} with respect to parameter \( k \) if there is an algorithm that solves it in \( O(n^{ck}) \) time, where \( n \) is the input size, and \( c \) is a constant. Note that, for a fixed value of \( k \), the time complexity is polynomial in \( n \). We say that a problem is \textit{fixed-parameter \( k_1 \)-tractable and \( k_2 \)-polynomial} if there exists an algorithm that solves it in \( O(f(k_1) \times n^{ck_2}) \) time for some function \( f \) and constant \( c \).
3.1.2 Fixed-Parameter Tractability of Reachability Under Separate Administration

This section presents parameterized complexity results for user-role reachability under the separate administration restriction. Our exposition is for policies without role hierarchy. Analysis of policies with role hierarchy can be reduced to analysis of policies without role hierarchy, by transforming the policy and goal to make the effects of inheritance explicit, as described in [39].

A role is negative (in a problem instance) if it appears negated in some precondition in the policy; other roles are non-negative. A role is positive if it appears positively (i.e., not negated) in some precondition in the policy or appears in the goal; other roles are called non-positive roles. A role that is both negative and positive is called mixed. This section shows that reachability analysis is fixed-parameter tractable with respect to the number of mixed roles. We prove this constructively, by giving a fixed-parameter tractable algorithm based on a reduction theorem that shows that it is safe to execute certain sequences of transitions atomically, i.e., as a single larger (composite) transition. In some ways, our reduction is a special case of Lipton’s reduction [29], but in another way, our reduction differs from Lipton’s reduction and its numerous successors. Those reductions justify treating given sequences of transitions (which appear in the control flow graph of the program) as atomic, i.e., as composite transitions. An ARBAC policy has no control flow, so our reduction itself defines composite transitions, and justifies using them instead of the original transitions. In addition, we prove fixed-parameter tractability for our algorithm. We are not aware of any similar complexity results in the literature for reductions or partial-order reductions, whose performance is usually evaluated in a purely empirical way.

The heart of our method is the definition of a reduced transition relation $\sim$ that takes larger steps than the original transition relation $\rightarrow$; specifically, a single step of $\sim$ may

---

\[\text{Our algorithm is not a special case of traditional partial-order reductions [11; 4], because our algorithm exploits the fact that certain transitions are left-movers, while traditional partial-order algorithms exploit only full commutativity (independence) of transitions.}\]
correspond to multiple steps of $\rightarrow$. The reachability algorithm itself is a straightforward exploration of the states reachable from the initial state via the reduced transition relation. Note that increasing the transition size by a factor of $k$ can reduce the number of explored states by a much greater factor, because it can eliminate many intermediate states produced by execution of different subsets of the original transitions that are aggregated into the reduced transitions.

An **invisible transition** is a transition that adds a non-negative role or revokes a non-positive role; other transitions are called **visible transitions**. The reduced transition relation differs from the original transition relation in two ways. (1) Transitions that revoke non-negative roles or add non-positive roles are prohibited; they are useless because they do not add roles in the goal and do not enable any transitions. (2) Invisible transitions get combined with a preceding visible transition to form a single composite transition. Invisible transitions can safely be executed immediately after the preceding visible transition, because they never disable any transitions.

More formally, for a state $\gamma$, let $\text{closure}(\gamma)$ denote the largest (with respect to $\subseteq$) state $\gamma'$ reachable from $\gamma$ by executing invisible transitions. A straightforward proof, based on the definition of invisible transition, shows that this closure is well-defined, i.e., there is a unique largest such state. The reduced transition relation is defined by: $\gamma_1 \xrightarrow{a} \gamma_2$ iff there exists a state $\gamma$ such that $\gamma_1 \xrightarrow{a} \gamma$ and $\gamma_2 = \text{closure}(\gamma)$ and $\gamma_1 \neq \gamma_2$ and $a$ is $\text{UserAssign}(r)$ or $\text{UserRevoke}(r)$ for some negative role $r$ (we don’t need to allow non-negative roles here, because they are added implicitly by closure and never revoked).

The following theorem shows that user-role reachability can be solved by exploring the reduced transition relation. For a problem instance $I$, let $PR_I$, $NR_I$, and $\overline{NR}_I$ denote the sets of positive, negative, and non-negative roles, respectively, in $I$. For a relation $\rightarrow$, let $\rightarrow^*$ denote its reflexive-transitive closure. A goal is reachable from a state $\gamma_0$ iff $(\exists \gamma : \gamma_0 \rightarrow^* \gamma \land \text{goal} \subseteq \gamma)$.

**Theorem 1** For all miniRBAC states $\gamma_0$ and all goals $\text{goal}$, $\text{goal}$ is reachable from $\gamma_0$ iff
\((\exists \gamma : \text{closure}(\gamma_0) \rightarrow^* \gamma \land \text{goal} \subseteq \gamma)\).

The proof is straightforward. This theorem implies that reachability analysis can be solved by computing the states reachable from \(\text{closure}(I)\) via \(\rightarrow\), and checking whether the goal is a subset of any of the resulting states. The graph constructed by this computation is called the reduced state graph.

**Theorem 2** The reduced state graph can be constructed in time \(O(f(|NR \cap PR|)|I|^c))\), for some function \(f\) and some constant \(c\). Thus, user-role reachability is fixed-parameter tractable with respect to the number of mixed roles.

**Proof:** To reduce clutter, we omit the subscript \(I\) on \(NR\) and \(PR\). We introduce some terminology. An \(NR\)-state is a subset of \(NR\). The \(NR\)-state graph is the projection of the reduced state graph onto \(NR\)-states; thus, an edge \((s_1, s_2)\) in the state graph induces an edge \((s_1 \cap NR, s_2 \cap NR)\) in the \(NR\)-state graph. Let \(G_{\text{red}}\) and \(G_{NR}\) denote the reduced state graph and the \(NR\)-state graph, respectively.

We show that the number of states in \(G_{\text{red}}\) is \(O(f(|NR \cap PR|))\) for some function \(f\). Every state in \(G_{\text{red}}\) is reachable by a simple path in \(G_{\text{red}}\). Every simple path in \(G_{\text{red}}\) corresponds, by projection, to a distinct path in \(G_{NR}\), because every \(\rightarrow\) transition changes the set of \(NR\) roles in the state. Furthermore, these paths in \(G_{NR}\) contain at most one occurrence of each cycle in \(G_{NR}\), because going around a cycle in \(G_{NR}\) a second time would not add any more \(NR\) roles to the state, hence the corresponding fragment of the path in \(G_{\text{red}}\) (note that a path in \(G_{NR}\) uniquely determines a corresponding path in \(G_{\text{red}}\), because \(\rightarrow\) adds all allowed positive roles at each step) would be a cycle, contradicting the assumption that the path in \(G_{\text{red}}\) is simple. Therefore, the number of states in \(G_{\text{red}}\) is bounded by the number of paths in \(G_{NR}\) that go around each cycle at most once. This number is clearly bounded by some function of the number of nodes in \(G_{NR}\). The number of nodes in \(G_{NR}\) is clearly bounded by some function of \(|NR|\). To see that it is bounded by some function of \(|NR \cap PR|\), note that the set of non-positive roles in the state is the same in every state in \(G_{\text{red}}\) except the initial state, because the reduced transition relation never adds non-positive.
roles to the state, and revocation has no preconditions, and hence invisible transitions that revoke non-positive roles occur only in composite transitions leaving the initial state. Thus, the set of non-positive roles is the same in every state in $G_{NR}$ except the initial state, so the number of nodes in $G_{NR}$ is bounded by some function of the number of positive roles in $NR$.

The time complexity of standard state-graph construction algorithms is polynomial in the size of the input and linear in the size of the output (i.e., the reduced state graph). Therefore, the worst-case time complexity of constructing the reduced state graph is $O(f(|NR \cap PR|)|I|^c)$, for some function $f$ and some constant $c$.

\[ \square \]

**Slicing.** Before applying the above algorithm, we apply a slicing transformation that back-chains along the rules to identify roles and rules relevant to the given goal, and then eliminates the irrelevant ones. The special twist here, compared to traditional cone-of-influence reduction [4], is to take into account whether a role appears positively or negatively. Let $ppre(t)$ be the set of roles used as positive preconditions in a `can_assign` rule $t$. Let $posPre = \{ (p,r) | \exists t \in can\_assign : p \in ppre(t) \land \text{target}(t) = r \}$. Define $Rel_+$ (“relevant positive roles”) by $Rel_+ = \{ p | (p,r) \in posPre^* \land r \in \text{goal} \}$. Note that $Rel_+$ contains every role $r$ such that adding $r$ to the state might be useful in reaching the goal. Roles that appear negatively in the preconditions of `can_assign` rules whose target is in $Rel_+$ are also relevant, so we define $Rel_-$ to contain those roles.

The sliced problem instance is obtained by deleting roles not in $Rel_+ \cup Rel_-$, deleting `can_assign` rules whose target is not in $Rel_+$, and deleting `can_revoke` rules whose target is not in $Rel_-$. Note that slicing can turn a negative role into a non-negative role, increasing the benefit of the reduction.

**Examples.**

**Example 1** Consider the following ARBAC policy $\psi$ and the reachability analysis problem instance $I = (\{r_1\}, \psi, \{r_5\})$. 
Figure 3.1: (a) Part of state space generated without reduction and without slicing (32 states, 96 transitions). (b) State space generated without reduction and with slicing (8 states, 16 transitions). (c) State space generated with reduction and without slicing (3 states, 3 transitions). ua and ur abbreviate UserAssign and UserRevoke, respectively. The state space generated with reduction and slicing (not shown) contains 1 state, namely, \{r_1, r_2, r_3\}, and 0 transitions.

1. \text{can\_assign}(true, r_1)
2. \text{can\_assign}(r_1, r_2)
3. \text{can\_assign}(r_2, r_3)
4. \text{can\_assign}(r_3, r_4)
5. \text{can\_assign}(r_4 \land \neg r_3, r_5)
6. \text{can\_assign}(r_2, r_6)
7. \text{can\_assign}(\neg r_1, r_7)

First, the algorithm performs slicing and computes $\text{Rel}_+ (I)$ and $\text{Rel}_- (I)$: $\text{Rel}_+ (I) = \{r_1, r_2, r_3, r_4, r_5\}$ and $\text{Rel}_- (I) = \{r_3\}$. $r_3$ is a mixed role, and $r_1, r_2$ and $r_4$ are roles
that are both positive and non-negative. All rules are relevant to the goal $\{r_5\}$ except rules 6 and 7. Next, the algorithm constructs the reduced transition graph $G(I)$ by computing $\text{closure}(\{r_1\}, I) = \{r_1, r_2\}$ and a set of states reachable from $\text{closure}(\{r_1\}, I)$. The graph $G(I)$ is given in Figure 3.2. The mixed role $r_3$ is added to states through visible transition. Roles $r_1$, $r_2$, 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.

![Figure 3.2: The reduced transition graph constructed in Example 1 by the forward algorithm.](image)

**Example 2** Consider the ARBAC policy

1. $\text{can}\_\text{assign}(r_1, r_2)$
2. $\text{can}\_\text{assign}(r_2, r_3)$
3. $\text{can}\_\text{assign}(r_3 \land \neg r_4, r_5)$
4. $\text{can}\_\text{assign}(r_5, r_6)$
5. $\text{can}\_\text{assign}(\neg r_2, r_7)$
6. $\text{can}\_\text{assign}(r_7, r_8)$

All roles can be revoked except $r_4$ and $r_8$

Consider the reachability problem for this policy with initial state $UA_0 = \{r_1, r_4, r_7\}$ and goal $= \{r_6\}$. The goal is not reachable from the initial state. Figure 3.1 describes the sets of reachable states and transitions generated using four variants of forward search, obtained by independently turning reduction and slicing on and off. For this policy, $NR = \{r_2, r_4\}$.

**Total Revocation and State Merging.** Total Revocation holds for a problem instance if every role that can be assigned by one of the administrators can also be revoked by one of the administrators, i.e., $(\forall (c, r) \in \text{can}\_\text{assign} : r \in \text{can}\_\text{revoke})$. In most ARBAC policies, an administrator who can assign users to a role can also revoke users from that role, and vice versa. Thus, in practice, most problem instances satisfy total revocation. For those problem instances, state merging can be used to optimize the above algorithm.

---

3 This simple definition is suitable when the separate administration restriction or hierarchical role assignment (cf. Section 3.2) holds.
Specifically, two reachable states \( s_1 \) and \( s_2 \) can be merged during the search if \((s_1 \cap NR) = (s_2 \cap NR)\) and \((s_1 \cap NR) \supseteq (UA_0 \cap NR)\), because a plan that reaches \( s_1 \cup s_2 \) from \( \gamma_0 \) can be constructed by concatenating (1) a plan that reaches \( s_1 \) from \( \gamma_0 \), (2) revocations of roles in \(((s_1 \cap NR_I) \setminus UA_0)\), and (3) a plan that reaches \( s_2 \) from \( \gamma_0 \). A corollary of this result is: for problem instances \( I \) that satisfy total revocation and \( UA_0 \cap NR_I \), two reachable states \( s_1 \) and \( s_2 \) can be merged if \( s_1 \cap NR = s_2 \cap NR \). This implies that, for such problem instances, the state graph has at most \( 2^{|NR|} \) nodes, i.e., the function \( f \) in Theorem 2 is \( f(x) = 2^x \) (in general, \( f \) could be larger than exponential).

### 3.1.3 Fixed-Parameter Tractability of Reachability With One Positive Precondition Under Separate Administration

The “one positive precondition” restriction, denoted \(|ppre| \leq 1\), means that the precondition of each \( can\_assign \) rule contains at most one positive literal. This section considers policy analysis under the \(|ppre| \leq 1\) and separate administration restrictions.

Sasturkar et al. showed that reachability for policies satisfying \(|ppre| \leq 1\), separate administration, \( \overline{CR} \) (all roles can be unconditionally revoked), and \( EN \) (no explicit negation, i.e., negation is used only in the form of static mutually exclusive roles constraints) is fixed-parameter polynomial with respect to the goal size [39].

We generalize that result by eliminating the restrictions on revocation and negation. This leads to the result that reachability for policies satisfying \(|ppre| \leq 1\) is fixed-parameter \(|Irrev|-tractable and \(|goal|-polynomial, where \( Irrev \) is the set of irrevocable roles. If we allow those parameters to vary, the worst-case running time is exponential in the goal size, and doubly exponential in the number of irrevocable roles. We believe that the algorithm is practical nevertheless, primarily because both parameters are very small (two or less) in all natural examples we have considered so far. Also, in experiments with synthetic examples with more irrevocable roles (see Section 3.5), the measured running time increases only modestly with \(|Irrev|\); we expect that the worst-case doubly-exponential behavior occurs only in contrived examples.
Let $I = (\gamma_0, \psi, \text{goal})$ be a problem instance satisfying $|ppre| \leq 1$, where $\gamma_0 = \langle R, UA_0 \rangle$ and $\psi = \langle \text{can_assign}, \text{can_revoke} \rangle$. Because $|ppre| \leq 1$, each element of $\text{can_assign}$ can be written in the form $\langle p \land \neg N, r \rangle$, where $p$ is a positive literal (i.e., a role) or $true$, and $N$ is a (possibly empty) set of roles; $\neg \{n_1, n_2, \ldots \}$ abbreviates $\neg n_1 \land \neg n_2 \land \ldots$. Let $\text{Irrev}$ be the set of irrevocable roles, i.e., $\text{Irrev} = R \setminus \text{can_revoke}$.

The algorithm has two stages.

The first stage uses backwards search from the goal to construct a directed graph $G = (V, E)$. The nodes correspond to states (i.e., sets of roles). The graph contains an edge from a state $UA_1$ to $UA_2$ if there is a $\text{can_assign}$ rule $(p \land \neg N, r)$ such that, starting from $UA_1$, revoking roles in $UA_1$ that appear in $N$ (we say that those roles “conflict” with the rule), and then adding $r$ using this rule leads to $UA_2$. Given $UA_2$, if such a predecessor state $UA_1$ exists, then we say that the rule is backwards enabled in $UA_2$. The predecessor function and backwards enabled function are defined as follows.

$$\text{pred}(\langle p \land \neg N, r \rangle, UA) = (p = true) \ ? \ UA \setminus \{r\} : (UA \setminus \{r\}) \cup \{p\}$$

$$\text{backEnab}(\langle p \land \neg N, r \rangle, UA) = r \in UA \land \text{pred}(\langle p \land \neg N, r \rangle, UA) \cap N = \emptyset$$

The graph is defined to be the least fixed-point of the following rules. The graph is computed by a straightforward workset algorithm.

$$\text{goal} \in V$$

$$(\forall UA_2 \in V, t \in \text{can_assign} : \text{backEnab}(t, UA_2) \Rightarrow e \in E \land \text{label}(e) = t \text{ where } e = \langle \text{pred}(t, UA_2), UA_2 \rangle)$$

$$(\forall \langle UA_1, UA_2 \rangle \in E : UA_1 \in V)$$

The second stage of the algorithm uses the graph $(V, E)$ to determine plan existence and, if the goal is reachable, produce a plan. The plan corresponds to a path in the graph from an initial node to the goal. However, each state encountered during the plan is not
simply (the set of roles in) the corresponding node in the path; rather, each state encountered in the plan may be a superset of the corresponding node in the path. This is because roles that were needed to satisfy preconditions of earlier transitions in the plan might still be in the state. This possibility is unavoidable, because some of those roles might be irrevocable. In addition, our algorithm leaves revocable roles in the state unless and until they need to be revoked to enable the next transition.

Let \( P = \langle e_1, e_2, \ldots, e_n \rangle \) be a path in \( \langle V, E \rangle \), represented as a sequence of edges. The candidate plan corresponding to \( P \), denoted \( \text{plan}(P) \), is \( A_1.A_2.\cdots.A_n \) where

- \( \langle p_i \land \neg N_i, r_i \rangle = \text{label}(e_i) \).
- \( UA'_i \), the intermediate state in the plan immediately before execution of \( A_i \), is defined by (1) \( UA'_1 \) is the source node of \( e_1 \), and (2) for \( i \geq 1 \), \( UA'_{i+1} = UA'_i \setminus \text{revoke}_i \cup \{r_i\} \).

Note that these intermediate states in the plan may be supersets of the corresponding nodes in the path, as discussed above.

- \( \text{revoke}_i = N_i \cap UA'_i \) (i.e., the set \( \text{revoke}_i \) of roles that need to be revoked by \( A_i \) contains roles that are present in the current state and conflict with the next \( \text{can_assign} \) transition).
- \( A_i = \{ \text{UserRevoke}(r) : r \in \text{revoke}_i \}.\text{UserAssign}(r_i) \).

Note that \( A_i \) consists of the indicated \( \text{UserRevoke} \) actions in arbitrary order, followed by the indicated \( \text{UserAssign} \) action.

We call \( \text{plan}(P) \) a “candidate plan” because it might attempt to revoke an irrevocable role. A path \( P \) is \( \text{feasible} \) if \( \text{plan}(P) \) does not contain \( \text{UserRevoke}(r) \) for any \( r \in \text{Irrev} \).

A node \( UA \) in the graph is an \( \text{initial node} \) if it is a subset of the initial state \( UA_0 \). This definition allows initial revocation of revocable roles in \( UA_0 \setminus UA \). This is necessary because every edge in the graph corresponds to a sequence of operations that adds at least one role to the state. Irrevocable roles in \( UA_0 \setminus UA \) are placed in \( \text{airs}(UA) \), defined below.

**Lemma 3** There is a plan for \( I \) iff there is a feasible path \( P \) from an initial node to the goal node.
To determine whether a feasible path exists, we compute, for each node \( UA \) in the graph, the set \( \text{airs}(UA) \) of sets of additional irrevocable roles that can be in states corresponding to that node; “additional” here means “not in \( UA \)”. More precisely, \( S \in \text{airs}(UA) \) iff (1) \( UA \) is an initial node and \( S = (UA_0 \cap \text{Irrev}) \setminus UA \), or (2) \( S \subseteq \text{Irrev} \) and \( S \cap UA = \emptyset \) and there exists a feasible path \( P \) from an initial node \( UA_1 \) to \( UA \) such that execution of \( \text{plan}(P) \) from \( UA_1 \) leads to the state \( UA \cup S \).

**Lemma 4** There is a plan for I iff \( \text{airs}(goal) \) is non-empty.

**Proof:** This is a corollary of the previous lemma, and the observation that (by definition of \( \text{airs} \)) \( \text{airs}(goal) \) is non-empty iff there exists a feasible path from an initial node to \( goal \). Note that \( \text{airs}(goal) \) might contain only the empty set. That counts! It implies there exists a feasible path \( P \) from an initial node to \( goal \) such that \( \text{plan}(P) \) adds no additional irrevocable roles.

Considering every path individually would be very expensive, so we introduce an alternate characterization of \( \text{airs} \) that leads to a more efficient algorithm. Specifically, \( \text{airs} \) is the least (with respect to the pointwise extension of the subset ordering) solution of the following set inclusion constraints, where the set comprehension \( \{ f(x_1, x_2, \ldots) : x_1 \in S_1, x_2 \in S_2, \ldots | p(x_1, x_2, \ldots) \} \) denotes the set obtained by iterating over each combination of values \( x_1, x_2, \ldots \) in \( S_1 \times S_2 \times \cdots \) and, if \( p(x_1, x_2, \ldots) \) holds, adding \( f(x_1, x_2, \ldots) \) to the result set.

- For each initial node \( UA \), \( \text{airs}(UA) \supseteq (UA_0 \cap \text{Irrev}) \setminus UA \).
- For each edge \( UA_1 \xrightarrow{(p \wedge \neg N, r)} UA_2 \), if \( UA_1 \) is reachable with additional irrevocable roles \( S \) in the state, and if this edge is a feasible transition from that state (i.e., if \( ((UA_1 \cap \text{Irrev}) \cup S) \cap N = \emptyset \)), then \( UA_2 \) is reachable with additional irrevocable roles \( S \cup ((UA_1 \setminus UA_2) \cap \text{Irrev}) \) (in other words, irrevocable roles present in \( UA_1 \) must still be present in the next state and, if they do not appear in \( UA_2 \), are “addi-
tional” in that state, by definition); formally,

\[
airs(UA_2) \supseteq \{ S \cup ((UA_1 \setminus UA_2) \cap \text{Irrev}) : S \in \ airs(UA_1) \mid ((UA_1 \cap \text{Irrev}) \cup S) \cap N = \emptyset \}
\]

The role \( r \) added along this edge does not appear in the set constraint, because the construction of the graph ensures that \( r \) appears in \( UA_2 \) and hence is never “additional” in states corresponding to \( UA_2 \). As an optimization, \( ((UA_1 \cap \text{Irrev}) \cup S) \cap N \) can be simplified to \( S \cap N \), because the construction of the graph implies \( UA_1 \cap N = \emptyset \). Similarly, the construction implies that \( (UA_1 \setminus UA_2) \) equals \( \{ p \} \) if \( p \neq \text{true} \) and equals \( \emptyset \) otherwise.

These constraints can be rewritten as a monotonic recursive definition.

\[
airs(UA_2) = (UA_2 \in \text{initialNodes} \Rightarrow \{ \emptyset \}) \cup \\
\{ S \cup ((UA_1 \setminus UA_2) \cap \text{Irrev}) : \\
UA_1 \xrightarrow{(N,r)} UA_2 \in \text{inedges}(UA_2), \\
S \in \ airs(UA_1) \mid ((UA_1 \cap \text{Irrev}) \cup S) \cap N = \emptyset \}
\]

The solution can be computed by a straightforward fixed-point computation, using a workset algorithm. Existence of a plan for \( I \) is then determined using Lemma 4. The fixed-point computation can easily be augmented to store additional information that provides a plan for \( I \), if a plan for \( I \) exists.

**Example 3** Consider the following ARBAC policy \( \psi' \) and the reachability analysis problem instance \( I' = (\{ r_4 \}, \psi', \{ r_5 \}) \).

1. \( \text{can_assign}(\text{true}, r_1) \)  
2. \( \text{can_assign}(r_1, r_2) \)  
3. \( \text{can_assign}(r_2, r_3) \)  
4. \( \text{can_assign}(r_3 \land \neg r_2, r_5) \)  
5. \( \text{can_assign}(r_4 \land \neg r_3, r_6) \)  
6. \( \text{can_assign}(r_6 \land \neg r_4, r_5) \)  
7. \( \text{can_assign}(r_2, r_7) \)

All roles are revocable

The graph computed in the first stage of the backward algorithm is given in Figure 3.3. 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, \( \text{airs}(\{r_4\}) = \emptyset \). The algorithm then computes \( \text{airs}(\{r_6\}) \) and \( \text{airs}(\{r_5\}) \), which are \( \emptyset \). Because \( \text{airs}(\{r_5\}) \) is not empty, the goal is reachable. Since the algorithm does not process \( \emptyset \), \( \text{airs}(\emptyset) = \text{airs}(\{r_1\}) = \text{airs}(\{r_2\}) = \text{airs}(\{r_3\}) = \emptyset \).

![Graph](image.png)

**Figure 3.3:** The graph constructed in Example 3 by the backward algorithm.

Now we analyze the algorithm’s time complexity. Let \( g = |\text{goal}| \). Each node in \( V \) contains at most \( g \) roles, because the search starts with the state \( \text{goal} \), and the definition of \( UA_1 \) ensures that \( |UA_1| \leq |UA_2| \). There are \( \binom{|R|}{g} \) sets containing exactly \( g \) roles, and each of these sets has \( 2^g \) subsets, so \( |V| = O(2^g \binom{|R|}{g}) \), which is \( O(2^g |R|^g) \). \( |E| = O(|V|^2) \) which is \( O(2^{2g} |R|^{2g}) \). Processing each possible node or edge takes \( O(|I|) \) time, so the running time of Stage 1 is \( O(|E||I|) \). In Stage 2, each node in \( V \) gets labeled with a set of subsets of \( Irrev \), and propagating each subset along each edge takes \( O(|I|) \) time, so the running time of Stage 2 is \( O(|E|^{2|Irrev|} |I|) \). Thus, the overall running time is \( O(2^{2g} |R|^{2g} 2^{2|Irrev|} |I|) \). This shows that reachability for problem instances with \( |ppre| \leq 1 \) is fixed-parameter \( |Irrev| \)-tractable and \( |\text{goal}| \)-polynomial.

This algorithm and complexity result can be extended to handle a class of policies that do not satisfy \( |ppre| \leq 1 \). Extending the algorithm is easy: interpret \( p \) as a set of positive preconditions, instead of a single positive precondition, and in the definition (3.1) of \( \text{pred} \), replace “ \( \cup \{p\} \) “ with “ \( \cup p \) “. The consequence is that generated states may contain more than \( |\text{goal}| \) roles. A policy is cycle free if the \( \text{posPre} \) relation (defined in Section 3.1.2) is acyclic. Consider any path \( P \) in the generated graph from a state \( UA_1 \) to a state \( UA_2 \), for a cycle free policy. Since the policy is cycle free, the number of edges in \( P \) labeled by a selected rule \( t \) is bounded by \( |UA_2| \). Note that \( |UA_1| \) is larger than \( |UA_2| \) due to uses of \( t \) by at most \( |ppre(t)| \times |UA_2| \). Thus, the maximum size of a state in the graph is bounded (loosely) by \( \pi \times |\text{goal}| \), where \( \pi = \Pi_{t \in \text{can_assign}} \max(1, |ppre(t)|) \). The complexity analysis
proceeds as above, except with $g$ replaced with $\pi \times |\text{goal}|$. This shows that the extended algorithm is fixed-parameter $|\text{Irrev}|$-tractable and $(\pi \times |\text{goal}|)$-polynomial, for cycle-free policies. Note that $\pi = 1$ when $|\text{ppre}| \leq 1$, so this result specializes to the previous result, although only for cycle-free policies.

**Partial-Order Reduction.** The graph construction can be optimized with a partial-order reduction [11; 4]. It is trickier than in Section 3.1.2, because the graph construction is a backward search, and the states corresponding to nodes in the graph are only partly determined during the backward search. A straightforward adaptation of the reduction in Section 3.1.2 is unsound. That reduction executes transitions whose target is a non-negative role as soon as they are enabled, hence those transitions appear early in the plan. A straightforward adaptation of it for backward search is to defer processing of such transitions whenever other transitions are backwards enabled; this also causes such transitions to appear early in the generated plan. However, this is unsound: it sometimes defers such transitions too much. For example, consider a reachability problem instance with $\text{can\_assign} = \{\langle r_0, r_1 \rangle, \langle \neg r_0, r_2 \rangle, \langle \text{true}, r_0 \rangle, \langle \neg r_2, r_3 \rangle\}$, $\text{can\_revoke} = \emptyset$, $\text{UA}_0 = \emptyset$, and $\text{goal} = \{r_1, r_2\}$. The technique proposed above defers processing of rules whose target is the non-negative role $r_1$, so only the rule that adds $r_2$ would be explored backwards from the goal state. Therefore, the algorithm would fail to find the plan for this problem instance, namely, the plan that adds $r_2$, then $r_0$, and then $r_1$.

To avoid this problem, our reduction takes a different approach: it identifies transitions that can be processed eagerly during the backward search, causing them to appear late in the resulting plan. A role $r$ is *backwards invisible* in a state $UA$ if every transition $t = \langle p \land \neg N, r \rangle$ with target $r$ satisfies $(p = \text{true} \lor p \in UA \lor p \notin NR_I) \land N \cap \text{Irrev} = \emptyset$, and at least one transition with target $r$ is backwards enabled in $UA_2$. The first conjunct in the formula ensures that backward execution from $UA$ of a transition with target $r$ does not disable backwards-enabled transitions with other targets (so the backward algorithm can process those transitions after processing a transition for $r$). The second conjunct in the
formula ensures that transitions for $r$ will not be disabled by irrevocable roles added to the state in stage 2.

To incorporate the reduction into the graph construction, modify the definition of $E$ so that, if some role $r$ in $UA_2$ is backward invisible in $UA_2$ and the stack proviso is satisfied (see below), then only can_assign rules for $r$ are explored from $UA_2$, otherwise all backward enabled can_assign rules are explored from $UA_2$. The stack proviso ensures that no transitions are completely ignored even when the state space contains cycles; it is satisfied if at least one transition backward explored from the current state leads to a state not on the DFS search stack [11, Chapter 6]. Note that slicing can increase the benefit of this reduction, by turning negative roles into non-negative roles.

This reduction is not a special case of traditional partial-order reductions [11; 4], because it uses an extra condition to deal with the fact that the search is split into two stages, and because it exploits the fact that the eagerly executed transitions are left-movers, while traditional partial-order algorithms exploit only full commutativity (independence) of transitions.

### 3.2 Beyond the Separate Administration Restriction

This section considers policy analysis without the separate administration restriction. First, we generalize the reduction-based algorithm in Section 3.1.2 to eliminate its dependence on this restriction. Second, we identify a condition under which policy analysis without the separate administration assumption can be reduced to policy analysis with the separate administration assumption.

#### 3.2.1 Fixed-Parameter Tractability of Reachability

Without the separate administration restriction, reachability analysis must consider plans that may contain administrative actions that change the role memberships of any user in $U_0$, not only the target user. To accommodate this, we describe how to generalize the reduction-based algorithm in Section 3.1.2 to track the role sets of multiple users. The worst-case time complexity is exponential in the number of those users. This demonstrates that reachability
analysis is fixed-parameter tractable with respect to the number of negative roles and $|U_0|$.

Generalizing the partial-order algorithm in this way is straightforward. We now deal with full user assignments $UA \subseteq U \times R$, not simplified user assignments $UA \subseteq R$. We strengthen the definition of enabledness of an action in a state $\gamma$ to require (in addition to the conditions in Section 2.2) that the administrative role $r_a$ (the first argument to the $UserAssign$ or $UserRevoke$ action) is a role of some user in $U_0$ in $\gamma$. The definitions of visible and invisible transitions and closure are unchanged. The reduced transition relation is defined by:

$\gamma_1 \xrightarrow{a} \gamma_2$ iff there exists a state $\gamma$ such that $\gamma_1 \xrightarrow{a} \gamma$ and $\gamma_2 = \text{closure}(\gamma)$ and $\gamma_1 \neq \gamma_2$ and $a$ is $UserAssign(r_a, u, r)$ or $UserRevoke(r_a, u, r)$ for some administrative role $r_a$, some user $u$ in $U_0$, and some negative role $r$. Theorem 1 still holds, provided we replace $\text{goal} \subseteq \gamma$ with $\text{goal} \subseteq \gamma(u_t)$, where $\gamma(u) = \{r \mid \langle u, r \rangle \in \gamma\}$.

The proof of fixed-parameter tractability of this algorithm with respect to the number of negative roles and $|U_0|$ is analogous to the proof of Theorem 2, except that each state in $G_{NR}$ is now a $|U_0|$-tuple of subsets of $NR$, so the size of $G_{NR}$ is bounded by a function of and $|NR|$ and $|U_0|$, so by the same argument as before, the size of $G_{\text{red}}$ is also bounded by some function of $|NR|$ and $|U_0|$, i.e., the size of $G_{\text{red}}$ is $O(f(|NR|, |U_0|))$ for some function $f$. It follows that the worst-case time complexity of constructing the reduced state graph is $O(f(|NR|, |U_0|)|I|^c)$, for some function $f$ and some constant $c$.

**Example 4** Consider the following ARBAC policy $\psi$ 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. $can\_assign(r_1, r_6 \land \neg r_3, r_4)$

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
Figure 3.4: Graph constructed in Example 4 by the forward algorithm without separate administration.

regular role (i.e. appears as a postcondition of rule 1).

First the algorithm performs slicing and computes $\text{Rel}_+ (I) = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ and $\text{Rel}_- (I) = \{r_3\}$. $r_3$ is a mixed role and all other roles are both positive and non-negative. All rules are relevant.

Next, the algorithm computes closure($\{(u_1, r_1), (u_2, r_2), (u_3, r_4), I\}$) and all states reachable from closure($\{(u_1, r_1), (u_2, r_2), (u_3, r_4), I\}$). The resulting graph is given in Figure 3.4. Role $r_3$ is added to the initial state through a visible transition. Role $r_5$ is then added to the resulting state through an invisible transition. Because the graph contains $(u_3, r_5)$, the goal is reachable.

3.2.2 Hierarchical Role Assignment

We say that an administrative role $r$ is “available” in a state if some user in $U_0$ is a member of $r$ in that state.

The separate administration restriction simplifies policy analysis primarily because it ensures that the set of available administrative roles does not change. Here, we identify a condition under which changes to the set of available administrative roles, although possible, are not useful (for reaching a goal) and hence can be ignored.

The condition ensures that, in the initial state, the users in $U_0$ already have all of the administrative permissions of administrative roles to which they could assign themselves. This is achieved by requiring that the users in $U_0$ are implicit or explicit members of all those administrative roles in the initial state.

A miniARBAC policy has hierarchical role assignment with respect to a set $A$ of administrative roles if: for all $\langle ar, c, r \rangle$ in can_assign, if $ar$ is in $A$ and $r$ is an administrative role, then $ar \succeq r$. This implies that a user who can assign users to $r$ is an implicit mem-

4Recall that the concept of administrative role is well-defined even without the separate administration
ber of \( r \). We call this property “hierarchical role assignment” because it relates the role hierarchy with the \( \text{can\_assign} \) relation.

**Theorem 5** Let \( I = (\gamma_0, \psi, U_0, u_t, \text{goal}) \) be a reachability problem instance. Let \( A \) be the available administrative roles in \( \gamma_0 \). If the miniARBAC policy \( \psi \) has hierarchical role assignment with respect to \( A \), then the goal is reachable iff the goal is reachable via a plan in which all assign and revoke actions act on the target user.

**Proof:**

It is not useful for a user \( u_1 \) in \( U_0 \) to assign a user \( u_2 \) other than \( u_t \) in \( U_0 \) to an administrative role \( ar \), because \( u_1 \) is already an implicit member of \( ar \), and making another user a member of \( r \) provides no additional administrative permissions to the group \( U_0 \) of users. Note that we allow assignment of administrative roles to \( u_t \), because such assignments can truthify the precondition of a \( \text{can\_assign} \) rule, and this might enable addition of \( u_t \) to a regular role in the goal. It is not useful to assign a user \( u_2 \) other than \( u_t \) to a regular role, or to revoke \( u_2 \) from any role, because the only potential benefit of such administrative actions would be to truthify the precondition of a rule allowing \( u_2 \) to be assigned to an administrative role, and we already showed that the latter role assignment would be useless.

\[ \square \]

Our analysis algorithms in Sections 3.1.2 and 3.1.3 exploit separate administration only to avoid considering administrative actions that act on users other than the target user. Therefore, Theorem 5 implies that those algorithms work correctly for problem instances that satisfy hierarchical role assignment.

### 3.3 Other Analysis Problems

This section presents algorithms for some other analysis problems. These algorithms use algorithms for user-role reachability as a subroutine, so our results from Sections 3.1.2–3.2 are useful here. These algorithms can invoke specialized algorithms for reachability when the latter algorithm’s restrictions are satisfied (e.g., \( |\text{ppre}| \leq 1 \)). It is straightforward to restriction.
obtain fixed-parameter tractability results for these algorithms, based on our complexity results for user-role reachability.

**Role containment.** Role containment problem instances have the form [28]: in every state reachable from a given initial state, is every member of role $r_1$ also a member of role $r_2$? This problem can be reduced to user-role reachability by adding a new role $r$ and adding a $can_assign$ rule with precondition $r_1 \land \neg r_2$ and target $r$. The containment property holds iff, for every user $u$, there is not a reachable state in which $u$ is a member of $r$. As an optimization, only users with distinct sets of initial roles need to be considered.

**Weakest Preconditions.** Weakest precondition queries return the minimal sets of initial role memberships of the target user for which a given reachability goal is achievable (for these queries, the initial state does not specify the initial roles of the target user). An example query is: what are the weakest preconditions for a user initially in DeptChair to assign the target user to HonorsStudent? For policies satisfying the conditions of the backward algorithm in Section 3.1.3, that algorithm can be modified to efficiently answer such queries. For each leaf node $UA$ in the graph, we compute $airs(goal)$ taking $UA$ as the only initial node, and if $airs(goal)$ is non-empty, then $UA$ is a weakest precondition, unless we find another one that is a subset of $UA$.

### 3.4 Case Studies

This section briefly describes the ARBAC policies that we used as case studies. Details of both policies are available from [44].

Our main case study is an ARBAC policy for selected aspects of a university. The policy includes rules for assignment of users to various student and employee roles. Student roles are undergraduate student, graduate student, teaching assistant, research assistant, grader, honors student, graduate student officer, graduate education committee (which has a student member), etc. Employee roles are admissions officer, assistant professor, dean, dean of admissions, department chair, facilities committee, graduate admissions committee, grad-
uate education committee, honors program director, president, professor, provost, etc. The role hierarchy includes the relationships President $\succeq$ Provost $\succeq$ Dean $\succeq$ DeptChair $\succeq$ Professor. Sample can_assign rules are: the honors program director can add undergraduates to the honors student role; the president can assign a professor who is not a department chair to the provost role.

The policy has some limitations. Our ARBAC framework does not support parameters (of role, permissions, etc.), so the policy is for a single department, a single class, etc. With a framework that does, we could easily add parameters (such as department name and class number) to the policy. We could analyze the resulting policy by instantiating the parameters with appropriate sets of values and then applying our current analysis algorithms; we plan to develop algorithms that handle parameters directly for efficiency. The current policy is only for user-role assignment; this seems like a good place to start, since the user assignment changes more often. Policies for administration of the permission-role assignment and the role hierarchy will be added later. Despite these limitations, this ARBAC policy is a substantial case study compared to others in the literature, as discussed in Chapter 7.

These limitations do not significantly affect the characteristics of the policy that guided this work, such as the size of preconditions. Experience with this policy guided our choice of complexity parameters and restrictions for the analysis algorithms in Sections 3.1.2–3.1.3 and suggested realistic values for some of the parameters used in random policy generation in Section 3.5. In particular, we note the following characteristics of the policy:

- Every can_assign rule has at most one positive precondition, so the analysis algorithm in Section 3.1.2 applies.

- The policy does not satisfy separate administration, but it has hierarchical role assignment with respect to most sets of administrative roles, so most reachability properties of the policy can be analyzed using the algorithms in Sections 3.1.2 and 3.1.3; the others can be analyzed using the algorithms in Section 3.2.

- About 1/3 of the roles are administrative (i.e., have at least one administrative per-
mission). After the transformation to eliminate role hierarchy (cf. first paragraph of Section 3.1.2), about 1/4 of the roles are negative, about 1/4 are mixed, and about 2/3 are positive. Since about 3/4 of the roles are non-negative, the reduction in Section 3.1.2 should be effective.

- Problem instances with hierarchical role assignment have at most two irrevocable roles, because admissions officers and the graduate admissions committee can accept but not expel students.

Formulating and checking properties of the policy helped uncover some flaws in it, for example, a place where we accidentally used Student instead of Undergrad, and places where we forgot to take role hierarchy into account, e.g., places where we were thinking of Provost and President as Faculty only, forgetting that these roles also inherit (indirectly) from Staff.

Here are sample user-role reachability problem instances (with answers!) for this policy. Can a user initially in DeptChair and a user initially in Undergrad reach a state in which the latter user is in HonorsStudent? Yes, because the user in DeptChair can assign himself to HonorsProgramDirector, and then assign the undergrad to HonorsStudent. Can a user initially in Provost and a user initially in DeptChair reach a state in which latter user is in Dean? No, because the rule for assignment to Dean has precondition Professor ∧ ¬DeptChair. The Provost can remove users from DeptChair, but the Provost cannot add users to Professor (only the President can do that).

A sample role containment problem instance is: Is TA (i.e., teaching assistant) contained in Grad (i.e., graduate student)? No. Although Grad is a precondition for assignment to TA, a user can be revoked from Grad while remaining in TA. (This example illustrates that preconditions are not necessarily invariants.)

Our second case study is an ARBAC policy for a health care institution, based on the policy in [9], extended with some aspects of the policy in [3]. This case study is smaller, but we note that it shares most of the above characteristics of the university policy. In particular,
every can_assign rule has at most one positive precondition, separate administration is not satisfied, and about 1/3 of roles are negative, and problem instances with hierarchical role assignment have at most one irrevocable role. One difference is less hierarchical role assignment (i.e., there are more sets of administrative roles for which the policy does not have hierarchical role assignment).

3.5 Experimental Results

This section presents the results of experiments done to evaluate the performance of the forward and backward reachability algorithms. The forward algorithm always uses the reduced transition relation; the backward algorithm includes the partial-order reduction only when stated explicitly. The algorithms were applied to the case studies in Section 3.4 and randomly generated ARBAC policies. Our algorithm for random policy generation has several parameters, allowing control over the number of roles, percentage of negative roles, percentage of irrevocable roles, average number of positive and negative preconditions per rule, average number of rules per target role, etc. Generally, we used parameter values (e.g., for number of roles) and distributions of values (e.g., for number of rules per target role) similar to those in the university policy as a baseline, and then varied selected parameters to explore the effect. Each data point reported for randomly generated policies is an average over 32 policies generated using the same parameter values. In many cases, the standard deviation is comparable in magnitude to the average; this suggests that statistical fluctuations cause the observed aberrations in the trends (e.g., there are fewer states and transitions for |R| = 300 than |R| = 200 in Table 3.1(b)) and suggests seeking additional parameters to control random policy generation more tightly. Running times were measured on a 2.8 GHz Pentium D with 1 GB RAM running Linux 2.6.20. All reported running times are measured in seconds.

**Case Studies.** Both policies satisfy the restriction |ppre| ≤ 1 and hence both forward and backward reachability algorithms are applicable for queries satisfying hierarchical role assignment. For a variety of reachability queries with |goal| ≤ 2, forward with slicing
Table 3.1: (a) Running time of forward algorithm vs. number of mixed roles with $|R| = 32$. (b) Running time of forward algorithm vs. number of roles with 5 mixed roles.

<table>
<thead>
<tr>
<th>Mixed</th>
<th>State</th>
<th>Trans</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td>18</td>
<td>0.07</td>
</tr>
<tr>
<td>5</td>
<td>33</td>
<td>122</td>
<td>0.32</td>
</tr>
<tr>
<td>7</td>
<td>187</td>
<td>960</td>
<td>2.60</td>
</tr>
<tr>
<td>9</td>
<td>350</td>
<td>2238</td>
<td>5.62</td>
</tr>
<tr>
<td>11</td>
<td>1404</td>
<td>10699</td>
<td>24.7</td>
</tr>
<tr>
<td>13</td>
<td>5593</td>
<td>50511</td>
<td>128.10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Roles</th>
<th>State</th>
<th>Trans</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>42</td>
<td>152</td>
<td>1.09</td>
</tr>
<tr>
<td>200</td>
<td>54</td>
<td>193</td>
<td>4.42</td>
</tr>
<tr>
<td>300</td>
<td>41</td>
<td>159</td>
<td>5.40</td>
</tr>
<tr>
<td>400</td>
<td>57</td>
<td>218</td>
<td>13.65</td>
</tr>
<tr>
<td>500</td>
<td>74</td>
<td>266</td>
<td>17.63</td>
</tr>
</tbody>
</table>

terminates in at most 0.01sec and backward with reduction terminates in at most 0.19sec.

For each goal, the forward algorithm generates at most 2 states and 1 transition, and the backward algorithm with reduction generates at most 5 nodes and 4 transitions.

**Evaluation of the Forward Algorithm.** Table 3.1(a) shows the number of explored states, number of explored transitions, and running time of the forward algorithm (without slicing) on randomly generated policies with varying number of mixed roles and with other parameters held constant. All three cost metrics grow quickly as a function of the number of mixed roles. Table 3.1(b) shows the performance of the forward algorithm (without slicing) on randomly generated policies where the number of roles varies (the number of rules varies with it, since average number of rules per role is held constant), and the number of mixed roles is held constant. In this case, the cost grows much more slowly.

Slicing significantly improves the typical performance of the forward algorithm, although it does not change the worst-case performance. For most policies, the algorithm explored just one or two states after slicing, terminating within 0.04 sec.

**Evaluation of the Backward Algorithm.** We evaluated the backward algorithm on randomly generated policies satisfying $|ppre| \leq 1$. Table 3.2(a) shows that the analysis cost grows quickly as a function of the size of the goal when other parameters are held constant.

Table 3.2(b) shows that the analysis cost grows very slowly as a function of the number of roles (and rules), when the percentage of irrevocable roles and the goal size are held constant (at 5% and 1, respectively).

The reduction technique for backward algorithm reduces the state space and the running
Table 3.2: Performance of backward algorithm for (a) varying goal size and (b) varying number of roles with $|Irrev|/|R| = 0.05$ and $|goal| = 1$.

time of policies used in Table 3.2(a) by 24% and 19% on the average, respectively. It does not affect the state space and running time of policies used in Table 3.2(b).

**Forward Algorithm with Slicing vs. Backward Algorithm.** We applied the forward algorithm with slicing to the same policies used for the experiments reported in Table 3.2(a). The average execution times were 0.04 sec, 0.38 sec, 0.87 sec and 1.08 sec when $|goal|$ is 1, 2, 3, and 4, respectively. Observe that the average execution time for the forward algorithm increases slightly with $|goal|$, while the corresponding increase in the execution time for the backward algorithm is much more significant.

Note that slicing does not change the worst-case complexity of the forward algorithm. When $|goal| = 1$ and $|Irrev| \leq 1$, the backward algorithm has better time complexity than the forward algorithm, except when $|NR| = 0$ and both algorithms have similar (polynomial) complexity. For a set of randomly-generated policies with $|goal| = 1$, $|Irrev| = 2$, $|R| = 50$, and $|NR|$ varying between $0.6|R|$ and $0.9|R|$, the backward algorithm is 11 to 30 times faster than the forward algorithm. We observe that when $|goal|$ and $|Irrev|$ are small and fixed, the backward algorithm is superior to the forward algorithm in terms of analysis time and the size of the explored state space.
4 User-Role Reachability Analysis of Evolving Administrative Role Based Access Control

4.1 User-Role Reachability Under Separate Administration

This section presents incremental algorithms for user-role reachability analysis of ARBAC without separate administration.

4.1.1 Incremental Forward Algorithms

We present three forward algorithms – IncFwd1, IncFwd2, and LazyInc – for incremental reachability analysis of ARBAC. We consider the following changes: (1) add a can_assign rule, (2) delete a can_assign rule, (3) add a can_revoke rule, and (4) delete 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 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 \( I = \langle UA_0, \psi, goal \rangle \) 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 [46]. Below, we describe the three incremental analysis algorithms in detail.

4.1.2 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) add a `can_assign` rule; (2) add a `can_revoke` rule; (3) delete a `can_assign` rule whose target role is non-positive or is irrelevant to the goal; and (4) delete a `can_revoke` rule whose target role is neither a mixed role nor a negative role in the initial state.

• If the analysis result of $I$ is false, then the following changes will not affect the result: (1) delete a `can_assign` rule; (2) delete a `can_revoke` rule; (3) add a `can_assign` rule whose target role is non-positive or is irrelevant to the goal; and (4) add a `can_revoke` rule whose target role is neither a mixed role nor a negative role in the initial state.

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.

4.1.3 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.

**Add a can_revoke rule** Suppose that `can_revoke(T)` is added to the policy $\psi$. Let $I_1 = \langle UA_0, \psi \cup \{can_revoke(T)\}, goal\rangle$. Figure 4.1 gives the pseudocode for constructing graph $G_{inc}(I_1)$ from $G(I)$.

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 $T$,
Pseudocode for adding $can\_revoke(T)$

adds a transition $ur(T)$ and marks new target state as "UnProcessed" (lines 5 – 18). Otherwise, if $T$ is in $UA_0$ 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 19–28). 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 29). 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 31).
If the goal is reachable, the algorithm calls function \textit{markUnproc} to mark states in \( G_{\text{inc}}(I_1) \), 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” (lines 9, 16, 26, 29). Otherwise, the algorithm processes all states marked as ”UnProcessed” using the non-incremental algorithm (line 33).

**Example 5** Consider the analysis problem instance in Example 1.

The goal is not reached. The transition graph \( G(I) \) computed by the non-incremental algorithm is given in Figure 3.2.

Suppose that rule \textit{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 \( \{r_1, r_2, r_3, r_4\} \xrightarrow{ur(r_3)} \{r_1, r_2, r_4, r_5\} \). Because state \( \{r_1, r_2, r_4, r_5\} \) contains the goal, the algorithm returns true. Figure 4.2 gives the resulting graph.

![Figure 4.2: The transition graph computed in Example 5 by the incremental algorithm.](image)

**Delete a can\_revoke rule** Suppose that \textit{can\_revoke}(T) is removed from \( \psi \). Let \( I_2 = \langle UA_0, \psi \setminus \{\textit{can\_revoke}(T)\}, \text{goal} \rangle \). Deleting this rule does not change the analysis result if \( T \) is not a relevant mixed role and is not a negative role in \( UA_0 \). Otherwise, the initial state may change and transitions that revoke \( T \) should be deleted. Figure 4.4 gives the pseudocode of the algorithm.

![Figure 4.3: Graphs computed in Example 6 using: (a) the non-incremental forward algorithm; (b) the incremental forward algorithm.](image)
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 $UA_0$ and is both negative and non-positive, then after deleting `can_revoke(T)`, $T$ should be added back to the initial state (lines 15-30). In this case, the algorithm computes a set $RT$ of roles that may be invalidated by $T$ (function `computeRT` on line 16). A role $r \in RT$ if (1) $T$ is a negative precondition of a `can_assign` rule whose target role is $r$ or (2) there exists a role $r' \in RT$ such that $r'$ is a positive precondition of a `can_assign` rule whose target role is $r$. If the number of relevant roles in $RT$, which are both positive and non-negative, is small (line 17), then for every transition $s \xrightarrow{\alpha} s_1$ in $G(I)$ and state $s' \in G_{inc}(I_2)$ computed from $s$, the algorithm computes transition $s' \xrightarrow{\alpha} s'_1$ by removing such roles that do not appear in $s'$ from $s_1$, and computing the closure of the resulting state (line 23). The algorithm also checks if transitions that add mixed roles in $RT$ 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 30). If the state containing the goal is deleted, the algorithm performs non-incremental analysis from states marked as "UnProcessed" (lines 32-33).

**Example 6** Consider the analysis problem instance in Example 3.

The goal is reachable. The transition graph computed by the non-incremental algorithm is given in Figure 4.3(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 $\{r_1, r_4, r_6\} \xrightarrow{ar(r_4)} \{r_1, r_5, r_6\}$ is not valid any more and hence is removed. As a result, the goal state $\{r_1, r_5, r_6\}$ becomes invalidated. Next, the algorithm performs non-incremental analysis from state $\{r_1, r_2, r_4, r_6\}$, and adds the following three transitions: $\{r_1, r_2, r_4, r_6\} \xrightarrow{ar(r_2)} \{r_1, r_3, r_4, r_6\}$, $\{r_1, r_2, r_3, r_4, r_6\}$, and $\{r_1, r_2, r_3, r_4, r_6\} \xrightarrow{ar(r_2)} \{r_1, r_3, r_4, r_5, r_6\}$.
\[ \text{init}' = \text{closure}(\text{InitRmI}_2(UA_0), I_2) \]

if \( \text{goal} \subseteq \text{init}' \) then add \( \text{init}' \) to \( G_{\text{inc}}(I_2) \); return true endif

if \( T \in \text{Rel}_+(I_2) \cap \text{Rel}_-(I_2) \) then

\[ W = \text{reached} = \{ \text{init} \} \]

while \( W \neq \emptyset \) do

Remove \( s \) from \( W \)

for \( s \xrightarrow{\alpha} s_1 \in G(I) \) do

if \( \alpha \neq \text{ur}(T) \) then

add \( s \xrightarrow{\alpha} s_1 \) to \( G_{\text{inc}}(I_2) \)

if \( \text{goal} \subseteq s_1 \) then markUnproc(\( W \cup \{s\} \)); return true endif

if \( s_1 \notin \text{reached} \) then \( W = W \cup \{s_1\} \); reached = reached \( \cup \{s_1\} \) endif

elseif \( \text{goal} \subseteq s_1 \) then markUnproc(\( \{s\} \)) endif

endfor

endwhile

else \( T \in (UA_0 \cap \text{Rel}_-(I_2)) \) then

\( R_T = \text{computeRT}(T, I_2) \)

if \( |R_T \cap (\text{Rel}_+(I_2) \cap \text{Rel}_-(I_2))| \) is small then

\( W_1 = \text{reached}_1 = \{ (\text{init}, \text{init}') \} \)

while \( W_1 \neq \emptyset \) do

Remove \((s, s')\) from \( W_1 \)

for \( s \xrightarrow{\alpha} s_1 \in G(I) \) do

if \( \alpha \) is enabled from \( s' \) then

\[ s'_1 = \text{closure}(\{s_1 \}\ \big\langle (R_T \cap (\text{Rel}_+(I_2) \cap \text{Rel}_-(I_2)) \ \cap \{s'\} \cup \{T\}), I_2) \rangle \]

add \( s' \xrightarrow{\alpha} s'_1 \) to \( G_{\text{inc}}(I_2) \)

if \( \text{goal} \subseteq s'_1 \) then markUnproc(\( \{s'_1 (s_1, s'_1) \in W \} \cup \{s'\} \)); return true endif

if \( (s_1, s'_1) \notin \text{reached}_1 \) then \( W_1 = W_1 \cup \{s_1, s'_1\} \); reached_1 = reached_1 \( \cup \{s_1, s'_1\} \) endif

elseif \( \text{goal} \subseteq s_1 \) then markUnproc(\( \{s'\} \)) endif

endfor

endwhile

else perform non-incremental analysis from \( \text{init}' \) endif

else return the analysis result of \( I \) endif

if \( G_{\text{inc}}(I_2) = \emptyset \) then add \( \text{init}' \) to \( G_{\text{inc}}(I_2) \); return false endif

process all states marked UnProcessed with non-incremental alg.

**Figure 4.4**: Pseudocode for deleting can_revoke(\( T \))

```
to the graph. 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 4.3(b).

**Add a can_assign rule** Suppose that can_assign(\( P \land \neg N, T \)) is added to the policy \( \psi \). Let \( I_3 = (UA_0, \psi \cup \{\text{can_assign}(P \land \neg N, T)\}, \text{goal} \). Figure 4.5 gives the pseudocode for constructing graph \( G_{\text{inc}}(I_3) \) from \( G(I) \).

If \( T \) is non-positive or is irrelevant to the goal, then adding this rule does not change the transition graph (line 1). Otherwise, the classification of roles may change: (1) An irrele-
```
1 if $T \not\in \text{Rel}_+(I)$ then return the analysis result of $I$
2 else
3 $\langle \text{Rel}_+(I_3), \text{Rel}_-(I_3) \rangle = \text{inc slicing}()$
4 $\text{init}' = \text{closure}((\text{InitRm}_{I_3}(U_{A_0}), I_3)$
5 if $\text{goal} \subseteq \text{init}'$ then add $\text{init}'$ to $G_{\text{inc}}(I_3)$; return true endif
6 $\text{RevRoles} = \langle \text{init}' \setminus \text{init} \rangle \cap (\text{Rel}_+(I_3) \cap \text{Rel}_-(I_3)) \setminus \text{Irrev}$
7 $\text{PosNonnegToMix} = ((\text{Rel}_+(I) \setminus \text{Rel}_-(I)) \cap (\text{Rel}_+(I_3) \cap \text{Rel}_-(I_3))
8 AddRoles = $init \cap \text{PosNonnegToMix}$
9 if $\text{AddRoles} \cup \text{RevRoles} = \emptyset$ then
10 $\langle \text{answer}, \text{lastState} \rangle = \text{addTransSeq}(\text{init}, \text{init}', \text{RevRoles}, \text{AddRoles})$
11 if $\text{answer} = \text{true}$ then return true else $W = \text{reached} = \{\langle \text{init}, \text{lastState} \rangle \}$ endif
12 else $W = \text{reached} = \{\langle \text{init}, \text{init}' \rangle \}$ endif
13 while $W \neq \emptyset$ do
14 Remove $(s, s')$ from $W$
15 for $s' \Rightarrow s_1 \in G(I)$ do
16 $\text{AddRoles} = s_1 \cap \text{PosNonnegToMix}$
17 if $\text{AddRoles} \neq \emptyset$ then
18 $\langle \text{answer}, s_1' \rangle = \text{addTransSeq}(s, s', \emptyset, \text{AddRoles})$
19 if $\text{answer} = \text{true}$ then markUnproc($\{s'_j | (s_j, s'_j) \in W \} \cup \{s'\}$); return true endif
20 else
21 $s_1' = \text{closure}(s_1 \cup (s' \setminus s), I_3)$
22 add $s' \Rightarrow s_1' \to G_{\text{inc}}(I_3)$
23 if $\text{goal} \subseteq s_1'$ then markUnproc($\{s'_j | (s_j, s'_j) \in W \} \cup \{s'\}$); return true endif
24 endif
25 if $(s_1, s_1') \not\in \text{reached}$ then reached = reached $\cup \{(s_1, s_1')\}$; $W = W \cup \{(s_1, s_1')\}$ endif
26 endfor
27 if $\text{addNewTrans}(s') = \text{true}$ then markUnproc($\{s'_j | (s_j, s'_j) \in W \} \cup \{s'\}$); return true endif
28 endwhile
29 if $G_{\text{inc}}(I_3) = \emptyset$ then add $\text{init}'$ to $G_{\text{inc}}(I_3)$; return false endif
30 process all states marked UnProcessed with non-incremental alg.
31 endif

![Figure 4.5: Pseudocode for adding can_assign($P \land \lnot N, T$)](image)

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

1. $\text{can assign}(P \land \lnot N, T) \in \text{RelRule}$. 
2. a $\text{can assign}$ rule $\rho \in \text{RelRule}$ if
   a. $\text{target}(\rho) \in \text{Rel}_+(I_3)$ and there exists $\rho' \in \text{RelRule}$ such that $\text{target}(\rho') \in \text{poscond}(\rho)$ or

39
(b) \textit{target}(\rho) is a positive role relevant to \textit{T}.

(3) $can\textunderscore revoke(r) \in RelRule$ if \( r \) is a mixed role in \( I_3 \) or is a negative role in \( UA_0 \).

\textit{RelRule} consists of (1) the new rule, (2a) relevant $can\textunderscore assign$ rules enabled by the new rule, (2b) $can\textunderscore assign$ rules that enable the new rule, and (3) $can\textunderscore revoke$ rules which revoke mixed roles or negative roles in \( UA_0 \).

Next, the algorithm computes the new initial state $init' = \text{closure}(InitRm_{I_3}(UA_0), I_3)$ (line 4), which may be different from the initial state $init = \text{closure}(InitRm_I(UA_0), I)$ of $G(I)$. Theorem 6 gives the relationship between $init'$ and $init$, which enables us to reuse $G(I)$ to construct $G_{inc}(I_3)$.

\textbf{Theorem 6} Let $init = \text{closure}(InitRm_I(UA_0), I)$ and $init' = \text{closure}(InitRm_{I_3}(UA_0), I_3)$.

One of the following holds: (1) $init = init'$; (2) $init' = \text{closure}(init, I_3)$; or (3) $init$ is reachable from $init'$ through a sequence of transitions: $init' \xrightarrow{ar(r_1)} s_1 \ldots \xrightarrow{ar(r_n)} s_n \xrightarrow{ua(r_{n+1})} s_{n+1} \ldots s_{m-1} \xrightarrow{ua(r_m)} \text{closure}(init \cup s_{m-1}, I_3)$ where \( \{r_1, \ldots, r_n\} \) are revocable mixed roles in $init' \setminus init$ and \( \{r_{n+1}, \ldots, r_m\} \) are roles in $init \setminus init'$ that are turned from both positive and non-negative to mixed.

\textbf{Proof:} Below, we prove each case of the theorem.

\textbf{Case 1:} $init' = init$ if no irrelevant roles in $UA_0$ become relevant, no relevant roles in $UA_0$ that are positive and non-negative become mixed, and no new invisible transitions are enabled from $init$ after the rule is added.

\textbf{Case 2:} If no irrelevant roles in $UA_0$ become relevant, no relevant roles in $UA_0$ that are positive and non-negative become mixed, and new invisible transitions are enabled from $init$ after the rule is added, $init'$ can be computed by executing all invisible transitions enabled from $InitRm_I(UA_0)$ in $I$ and then executing all invisible transitions that are enabled in $I_3$. Thus, $init' = \text{closure}(init, I_3)$. 

40
Case 3: We first prove the following Lemma and then use this lemma to prove the theorem.

**Lemma 7** Let $\text{init} = \text{closure}(\text{InitRm}_I(UA_0), I)$ and $\text{init}' = \text{closure}(\text{InitRm}_{I_3}(UA_0), I_3)$. Also, let $\{r_1, \ldots, r_m\}$ be roles in $\text{init} \setminus \text{init}'$ that are turned from both positive and non-negative to mixed. If $\text{init}' \setminus \text{init}$ does not contain revocable mixed roles, then $\text{init}$ is reachable from $\text{init}'$ through the following sequence of transitions: $\text{init}' \xrightarrow{ua(r_1)} s_1 \xrightarrow{ua(r_2)} \ldots \xrightarrow{ua(r_m)} \text{closure}(\text{init} \cup s_{m-1}, I_3)$.

**Proof:** We prove the Lemma by induction on the length of the sequence of transitions.

**Base Case:** If $\text{init} \setminus \text{init}'$ does not contain mixed roles, then either case (1) or case (2) in Theorem 6 applies.

**Induction:** Assume that the Lemma holds for $m = k - 1$. We prove the Lemma for $m = k$.

Assume that $\text{init} \setminus \text{init}'$ contains mixed roles $\{r_1, \ldots, r_k\}$. Such roles are turned from both positive and nonnegative to mixed after the can_assign rule is added, and were added to $\text{init}$ through invisible transitions. Since adding a can_assign rule does not disable transitions, we can perform transition $\text{init}' \xrightarrow{ua(r_1)} s_1$. Since this transition adds only $r_1$ and non-mixed roles to $s_1$, $\text{init} \setminus s_1$ contains mixed roles $\{r_2, \ldots, r_k\}$. By induction hypothesis, $\text{init}$ can be reached from $s_1$ through the sequence of transitions $s_1 \xrightarrow{ua(r_1)} \ldots \xrightarrow{ua(r_k)} \text{closure}(\text{init} \cup s_{k-1})$. Therefore, the lemma holds.

End-of-lemma □

Below, we prove the theorem by induction on $n$.

**Base Case:** If $\text{init}' \setminus \text{init}$ does not contain roles that changed from irrelevant to revocable mixed roles (i.e., $n = 0$), then Lemma 7 applies.

**Induction:** Assume that Case (3) of Theorem 6 holds for $n = k - 1$. We prove Case (3) for $n = k$. 

41
Assume that \( \text{init}' \setminus \text{init} \) contains mixed revocable roles \( \{r_1, \ldots, r_k\} \). Since adding \text{can}\_\text{assign} rule does not disable transitions, we can perform transition \( \text{init}' \xrightarrow{ur(r_1)} s_1 \). This transition revokes only role \( r_1 \) from \( \text{init}' \) and adds only non-mixed roles. As a result, \( s_1 \setminus \text{init} \) contains mixed revocable roles \( \{r_2, \ldots, r_k\} \). By induction hypothesis, \( \text{init} \) is reachable from \( s_1 \) through a sequence of transitions \( s_1 \xrightarrow{ur(r_2)} \ldots \xrightarrow{ur(r_k)} \text{closure}(\text{init} \cup \text{closure}(s_{k-1} \setminus \{r_k\}, I_3), I_3) \). Thus, Case (3) of Theorem 6 holds.

\( \square \)

Case (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 \( \text{init}' \setminus \text{init} \) to \( \text{init} \) and updates the graph (lines 20–24). In case (3), some roles in \( \text{init} \) are turned from both positive and non-negative to mixed (\text{AddRoles} in line 8), from irrelevant to relevant, or from revocable non-positive to mixed (\text{RevRoles} in line 6). In this case, the algorithm calls function \text{addTransSeq} (line 10) to add a sequence of transitions from \( \text{init}' \) to \( \text{closure}(\text{init} \cup s_{m-1}, I_3) \), which revokes roles in \text{RevRoles}, adds roles in \text{AddRoles}, and marks new states not containing the goal as "UnProcessed". This function returns \( \langle \text{answer}, \text{lastState} \rangle \) where \( \text{answer} \) is true if the sequence contains the goal state and is false otherwise, and \( \text{lastState} \) is the last state of the sequence.

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

Consider the following example:

\textbf{Example 7} Consider the ARBAC policy \( \psi \) in Example 1 and the reachability analysis problem instance \( I = \langle \emptyset, \psi, \{r_5\} \rangle \).

The transition graph computed by the non-incremental algorithm is given in Figure 3.2. The goal is not reached.

Suppose that \text{can}\_\text{assign}(r_3 \land \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. RelRule consists of rules 1-3, can_assign($r_3 \land \neg r_1, r_5$), and can_revoke($r_1$). The new initial state is $\emptyset$. Next, the algorithm applies Case (3) of Theorem 6 and adds transition $\emptyset \xrightarrow{ua(r_1)} \{r_1, r_2\}$ to the graph. The algorithm also adds transition $\{r_1, r_2\} \xrightarrow{ur(r_1)} \{r_2\}$ to the graph and marks states $\emptyset$ and $\{r_2\}$ as "UnProcessed".

Next, the algorithm processes transition $\{r_1, r_2\} \xrightarrow{ua(r_3)} \{r_1, r_2, r_3, r_4\}$. 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 $\{r_1, r_2, r_3, r_4\} \xrightarrow{ur(r_1)} \{r_2, r_3, r_4, r_5\}$ 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 4.6.

![Figure 4.6: The transition graph computed in Example 7 by the incremental algorithm.](image)

**Delete a can_assign rule** Suppose that can_assign($P \land \neg N, T$) is deleted from policy $\psi$. Let $I_4 = \langle UA_0, \psi \setminus \{\text{can_assign}(P \land \neg N, T)\}, \text{goal} \rangle$. If $T$ is not a positive role relevant to the goal, then the transition graph remains the same. 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($P \land \neg N, T$) 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). Below, we describe the algorithm and Figure 4.7 gives the pseudocode.
First, the algorithm performs slicing (function \textit{inc\_slicing} on line 2) and computes the new initial state \( \text{init}' = \text{closure}(\text{InitRm}_{I_4}(UA_0), I_4) \) (line 3). \( \text{init}' \) may be different from the initial state \( \text{init} = \text{closure}(\text{InitRm}_{I}(UA_0), I) \) of \( G(I) \). Theorem 8 gives the relationship between \( \text{init} \) and \( \text{init}' \), which enables us to reuse \( G(I) \) to construct \( G_{\text{inc}}(I_4) \).

\textbf{Theorem 8} Let \( \text{init} = \text{closure}(\text{InitRm}_{I}(UA_0), I) \), \( \text{init}' = \text{closure}(\text{InitRm}_{I_4}(UA_0), I_4) \), and \( \text{Invalid} = (\text{init} \setminus \text{init}') \cap (A_T \cup \{T\}) \). One of the following holds: (1) \( \text{init}' = \text{init} \); (2) \( \text{init}' = \text{init} \setminus (\text{RevtoIrr} \cup \text{Invalid} \cup \{S \in \text{MixtoNonpos} | S \text{ is revocable}\}) \); or (3) \( G(I) \) contains the following sequence of transitions: \( \text{init} \xrightarrow{\text{ua}(r_1)} s_1 \ldots s_{n-1} \xrightarrow{\text{ua}(r_n)} (\text{init}' \cup (s_{n-1} \cap (\text{RevtoIrr} \cup \text{Invalid} \cup \text{MixtoNonpos}))) \) where \( \{r_1, \ldots, r_n\} = (\text{init}' \setminus \text{init}) \cap \text{MixtoNonneg} \). \( \Box \)

\textbf{Proof:} We prove each case of Theorem 8 as follows.

\textbf{Case 1:} If no relevant roles in \( UA_0 \) become irrelevant, no mixed roles in \( UA_0 \) turn to both revocable and non-positive, no mixed roles reachable from \( \text{init} \) in \( I_4 \) become both positive and non-negative, and no invisible transitions in \( I \) that add roles to \( \text{InitRm}_{I}(UA_0) \) become disabled after the rule is deleted, then \( \text{init}' = \text{init} \).

\textbf{Case 2:} If no mixed roles reachable from \( \text{init} \) in \( I_4 \) become both positive and non-negative, but \( UA_0 \) contains roles that turned from relevant to irrelevant, \( UA_0 \) contains roles that from mixed to both revocable and non-positive, or some invisible transitions in \( I \) that add roles to \( \text{InitRm}_{I}(UA_0) \) become disabled without the deleted rule, then \( \text{init}' \) can be computed by removing all such roles from \( \text{init} \). Thus, \( \text{init}' = \text{init} \setminus (\text{RevtoIrr} \cup \text{Invalid} \cup \{S \in \text{MixtoNonpos} | S \text{ is revocable}\}) \).

\textbf{Case 3:} Prove by induction on the length of the sequence of transitions.

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

\textit{Induction:} Assume that case (3) of Theorem 8 holds for \( n = k - 1 \) and prove case for \( n = k \).

Let \( \{r_1, \ldots, r_k\} \) represent the roles in \( \text{init}' \setminus \text{init} \) that turned from mixed to both positive and non-negative. Such roles were added to \( \text{init} \) through visible transitions. We can perform
transition \( \text{init} \xrightarrow{\text{ua}(r_1)} s_1 \) because it is enabled without the deleted \text{can\_assign} rule. This transition only adds \( r_1 \) and non-mixed roles to \( s_1 \). As a result, \( \text{init}' \setminus s_1 \) contains both positive and non-negative roles \( \{r_2, \ldots, r_k\} \). By induction hypothesis, \( \text{init}' \) can be reached from \( s_1 \) through a sequence of transitions \( s_1 \ldots s_{k-1} \xrightarrow{\text{ua}(r_k)} (\text{init}' \cup (s_{k-1} \cap (\text{RevtoIrr} \cup \text{Invalid} \cup \text{MixtoNonpos}))) \). Thus, case (3) of Theorem 8 holds.

\[ \square \]

Case (1) states that the initial state does not change. In Case (2), \( \text{init} \) 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 \text{can\_assign} rule is deleted. In this case, the algorithm updates the graph from \( \text{init} \), removes such roles from \( \text{init} \) and states reachable from \( \text{init} \), and removes transitions that add or revoke such roles. In case (3), \( \text{init} \) contains roles turned from mixed to both positive and non-negative. In this case, the algorithm identifies the state \( (\text{init}' \cup (s_{n-1} \cap (\text{RevtoIrr} \cup \text{Invalid} \cup \text{MixtoNonpos}))) \) in \( G(I) \). The algorithm then updates the graph from this state by removing roles in \( (s_{n-1} \cap (\text{RevtoIrr} \cup \text{Invalid} \cup \text{MixtoNonpos})) \) from this state and all reachable states.

The graph is updated as follows. First, the algorithm computes set \( A_T \) (function \text{computeAT} on line 8). Next, for every transition \( s_1 \xrightarrow{\alpha} s_2 \) in \( G(I) \), if \( \alpha \) adds/revokes a role that is turned from mixed to non-positive or if \( \alpha \) can no longer be derived, the algorithm removes the transition (lines 17-19). Otherwise, if \( s_2 \setminus s_1 \) contains \( T \) and \( T \) cannot be re-derived, the algorithm removes \( T \) and all roles that cannot be derived without \( T \) from \( s_2 \) (lines 21-22). If \( \alpha \) 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 (3) of Theorem 8 (function \text{findTransSeq} on line 28 which applies case (3) of Theorem 8 and returns the last state of the sequence).

**Example 8** Consider the analysis problem instance in Example 3. The goal is reachable. The transition graph computed by the non-incremental algorithm is given in Figure 4.3(a).
Suppose that Rule 6 is removed from the policy. First, the algorithm computes $\text{RevtoIrr} = \{r_4, r_6\}$, $\text{MixtoNonneg} = \{r_3\}$, and $\text{MixtoNonpos} = \{\}$. 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 $\{r_1, r_4, r_6\} \xrightarrow{\text{ua}(r_2)} \{r_1, r_2, r_4, r_6\}$: it removes $r_4$ and $r_6$ from both states, and adds $r_3$ to the target state. Finally, the algorithm removes transition $\{r_1, r_4, r_6\} \xrightarrow{\text{ur}(r_4)} \{r_1, r_5, r_6\}$, 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 $\{r_1, r_2, r_3\} \xrightarrow{\text{ur}(r_2)} \{r_1, r_3, r_5\}$. Because $\{r_1, r_3, r_5\}$ contains the
goal, the algorithm returns true. The resulting graph is given in Figure 4.8.

### 4.1.4 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. Due to space constraints, this section presents 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 $I = \langle UA_0, \psi, goal \rangle$ be a reachability analysis problem instance. The algorithm is described below.

**Add a can_assign or a can_revoke rule** Adding a can_assign or a can_revoke rule does not affect the analysis result though it may affect the transition graph. 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.

**Delete a can_assign or a can_revoke rule** Assume that $can_assign(P \land \neg N, T)$ is deleted from $\psi$. If $T$ is not a positive role relevant to the goal, the algorithm returns true. Otherwise, the algorithm performs the following steps.

Let $\psi' = (\psi \setminus \{can_assign(P \land \neg N, T)\}) \cup DelayedRule$ and $I' = \langle UA_0, \psi', goal \rangle$. First, the algorithm computes $Rel_+(I')$ and $Rel_-(I')$. The algorithm then updates the transition graph using the deleted rule and delayed operations that may affect the analysis result after the rule is deleted. Such operations include addition of can_assign rules in $DelayedRule$ whose target roles are in $Rel_+(I')$, and addition of can_revoke rules in $DelayedRule$ that revoke relevant mixed roles or negative roles in $UA_0$. Finally, the algorithm updates 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 sections 4.1.3 with the following difference: when applying Theorem 6, 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, the algorithm first considers the operation that deletes `can_assign(P \land \neg N, T)` and applies 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 4.1 and 4.5, 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 adopted the first approach.

**Example 9** Consider the analysis problem instance in Example 3.

The goal is reached. The transition graph computed by the non-incremental algorithm is given in Figure 4.3.

Assume that the following sequence of changes is made to the policy: (1) delete rule 7, (2) add `can_assign(r_4 \land \neg r_5, r_1)`, (3) add `can_assign(r_1, r_3)`, (4) delete `can_revoke(r_6)`, and (5) delete rule 3. The 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 \land \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 $UA_0$, 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. Upon this change, 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) turns role $r_2$ from both positive and non-negative to irrelevant, transition $\{r_1, r_4, r_6\} \xrightarrow{ua(r_2)} \{r_1, r_2, r_4, r_6\}$ is deleted. The algorithm then processes transition $\{r_1, r_4, r_6\} \xrightarrow{ur(r_4)} \{r_1, r_5, r_6\}$. Because $r_5$ becomes mixed, the algorithm applies Case (3) of Theorem 6 and adds transitions $\{r_1, r_4, r_6\} \xrightarrow{ur(r_4)} \{r_1, r_6\}$ and $\{r_1, r_6\} \xrightarrow{ua(r_5)} \{r_1, r_5, r_6\}$ to the graph. The goal is reached.

Deleting a `can_revoke` rule is handled similarly.

### 4.1.5 Incremental Backward Algorithm

This section presents a backward algorithm for incremental user-role reachability analysis. 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.

**Add a can_revoke rule** Assume that `can_revoke(T)` is added to the policy. Graph $G_b$ is unaffected and we simply update the airs of nodes from the first initial node $V_0$. Let $rm(T, airs(V))$ denote $\{S - \{T\}|S \in airs(V)\}$. $airs'(V)$ is computed as follows:

- $airs'(V_0) = rm(T, airs(V_0))$
• For every edge \( V_1 \xrightarrow{(P \land \neg N, r)} V_2 \), \( \text{airs}'(V_2) \) is computed as the union of the following sets:

1. \( \text{rm}(T, \text{airs}(V_2)) \)
   \( \left\{ S \cup (\text{Irrev} \cap (V_2 \setminus V_1)) \mid S \in \text{airs}'(V_1) \setminus \text{rm}(T, \text{airs}(V_1)) \right\} \)

2. \( \left\{ S \cup (\text{Irrev} \cap (V_2 \setminus V_1)) \mid S \in \text{airs}(V_1) \setminus \text{rm}(T, \text{airs}(V_1)) \right\} \)

3. \( ((T \in N) \Rightarrow \left\{ S \cup (\text{Irrev} \cap (V_2 \setminus V_1)) \mid S \in \text{airs}(V_1), T \in S, ((\text{Irrev} \cap V_1) \cup (S \setminus \{T\})) \cap N = \emptyset \right\}) : \emptyset \)

(1) contains \( \text{airs}(V_2) \) 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 \( \text{airs}(V_1) \) that did not satisfy the negative precondition of the edge because they contained \( T \); since \( T \) becomes revocable, they are added to \( \text{airs}'(V_2) \).

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

The goal is not reachable. Figure 4.9 gives the graph constructed in the first stage. There are two initial nodes: \( \emptyset \) and \( \{r_1\} \). \( \text{airs}(\emptyset) = \text{airs}(\{r_1\}) = \text{airs}(\{r_2\}) = \emptyset \), \( \text{airs}(\{r_3\}) = \{\{r_2\}\} \), \( \text{airs}(\{r_4\}) = \{\{r_2, r_3\}\} \), and \( \text{airs}(\{r_5\}) = \emptyset \).

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 \( \text{airs}(\emptyset) \) does not contain \( r_3 \), \( \text{airs}'(\emptyset) = \{\emptyset\} \). 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 \( \text{airs}'(\{r_4\}) = \{\{r_2\}\} \). Finally, the algorithm updates the airs of \( \{r_5\} \), which results in \( \text{airs}'(\{r_5\}) = \{\{r_2, r_4\}\} \). Therefore, the goal is reachable.

**Delete a can_revoke rule** Suppose that can_revoke\((T)\) is deleted from the policy. Graph \( G_b \) remains the same and we simply update the airs of nodes. First, starting from the first initial node \( V_0 \), the algorithm searches \( G_b \) along edges marked 1, for nodes whose airs may
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 $V' \xrightarrow{\alpha} V$ 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 $e = V_1 \xrightarrow{\alpha} V_2$, 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 up another initial node and repeats the above process.

**Example 11** Consider the analysis problem instance in Example 3.

Figure 3 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 state 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 up another initial node $\emptyset$ and computes the airs of nodes reachable from $\emptyset$ using the non-incremental algorithm, which results in $airs'(\{r_5\}) = \emptyset$. Hence, the goal is reachable.

**Add a can_assign rule** Suppose that $can\_assign(P \land \neg N, T)$ 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 $V \xrightarrow{\alpha} V'$, 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
If the previous analysis result is true, the algorithm returns true. Otherwise, the 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 $V_1 \xrightarrow{\alpha} V_2$, 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 12** Consider the analysis problem in Example 1. The goal is not reachable. Figure 4.9 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 $\{r_1\} \xrightarrow{(r_1,r_4)} \{r_4\}$ 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.

**Delete a can_assign rule** Suppose that $can\_assign(P \land \neg N, T)$ is deleted from the policy. If $T$ is not a positive role relevant to the goal, the algorithm returns the previous 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 $V_1 \xrightarrow{\alpha} V_2$, if $\alpha \neq \langle P \land \neg N, T \rangle$ 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 $V_1 \xrightarrow{(P \land \neg N, T)} V_2$, 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 $V_1 \rightarrow V_2$. Finally, the algorithm updates the airs of nodes reachable from nodes in $L_T$: for every edge $V_1 \rightarrow V_2$, $airs'(V_2)$ 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 13** Consider the analysis problem in Example 3.

The graph computed by the non-incremental algorithm is shown in Figure 3. The 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 $can\_assign(r_6 \land \neg r_4, r_5)$ 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 $\{r_6\} \rightarrow \{r_5\}$ 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 $\{r_6\} \rightarrow \{r_5\}$ and $\{r_4\} \rightarrow \{r_6\}$. Because $airs'(\{r_5\})$ becomes empty, the algorithm picks up 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 shown in Figure 4.10.
4.1.6 Experimental Results Under Separate Administration

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

4.1.7 Experimental Results: Incremental Forward Analysis Algorithms

We apply the non-incremental and incremental forward algorithms to an ARBAC policy $\psi_1$ generated using the random policy generation algorithm in [46]. The parameter values (e.g., the percentage of mixed roles) in $\psi_1$ are similar to those in the university ARBAC policy developed in [46]. We choose two goals $\text{goal}_1$ and $\text{goal}_2$ such that $\text{goal}_1$ is reachable from the initial state $\emptyset$ and $\text{goal}_2$ is unreachable, and the size of transition graphs constructed during analysis is reasonably large (155478 transitions for $\text{goal}_1$ and 225792 transitions for $\text{goal}_2$). We then randomly generate a set of operations that add rules to $\psi_1$ or delete rules from $\psi_1$, and use them to compare the performance of our incremental algorithms against the non-incremental algorithm.

Table 4.1 compares the execution time of the non-incremental algorithm without saving the graph (NonInc)\(^5\), IncFwd1, and IncFwd2 for goals $\text{goal}_1$ and $\text{goal}_2$, when a change is made to policy $\psi_1$. Each 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 $\psi_1$, 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 4.1 shows that, when a can_revoke rule is added, IncFwd2 is 68.27 and 9.94 times faster than NonInc for $\text{goal}_1$ and $\text{goal}_2$, respectively. When only the 2 rules that revoke mixed roles are considered, IncFwd2 is

\(^5\)The overhead for saving the transition graph is 0.11 seconds – 0.27 seconds.
Table 4.1: Performance results of NonInc, IncFwd1, and IncFwd2 on $\psi_1$ for goals $goal_1$ and $goal_2$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>States</th>
<th>Trans</th>
<th>$goal_1$</th>
<th>States</th>
<th>Trans</th>
<th>$goal_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Time(Sec.)</td>
<td></td>
<td></td>
<td>Time(Sec.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NonInc</td>
<td>IncFwd1</td>
<td>IncFwd2</td>
<td>NonInc</td>
</tr>
<tr>
<td>add can_assign</td>
<td>30567</td>
<td>233299</td>
<td>15.68</td>
<td>0</td>
<td>5.87</td>
<td>33936</td>
</tr>
<tr>
<td>delete can_assign</td>
<td>19983</td>
<td>153931</td>
<td>47.41</td>
<td>45.90</td>
<td>7.08</td>
<td>18000</td>
</tr>
<tr>
<td>add can_revoke</td>
<td>20866</td>
<td>159290</td>
<td>49.43</td>
<td>0</td>
<td>0.72</td>
<td>19968</td>
</tr>
<tr>
<td>delete can_revoke</td>
<td>20297</td>
<td>154853</td>
<td>47.96</td>
<td>11.87</td>
<td>0.56</td>
<td>18432</td>
</tr>
</tbody>
</table>

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.

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.

**Results for adding/deleting a $can_assign$ rule** All $can_assign$ rules added/deleted are relevant to the goal. Observe from Table 4.1 that, when a $can_assign$ rule is added, IncFwd2 is 13.91 and 1.63 times faster than NonInc for $goal_1$ and $goal_2$, 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 (1165sec vs 1199sec). When considering both goals, IncFwd2 is 2.32 and 1.53 times faster than NonInc and IncFwd1, 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 2.49 times faster than NonInc and IncFwd1, respectively.
<table>
<thead>
<tr>
<th>Goal</th>
<th>States</th>
<th>Trans</th>
<th>Time(Sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>NonInc</td>
</tr>
<tr>
<td>$goal_1$</td>
<td>30219</td>
<td>240802</td>
<td>84.40</td>
</tr>
<tr>
<td>$goal_2$</td>
<td>18613</td>
<td>228819</td>
<td>74.00</td>
</tr>
</tbody>
</table>

Table 4.2: Performance results of NonInc, IncFwd1, IncFwd2, and LazyInc for $goal_1$ and $goal_2$ on ten sequences of operations.

**Results for adding/deleting a sequence of rules** Table 4.2 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 $\psi_1$ 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 $\psi_1$ with $goal_2$, for each operation, LazyInc is 16.08, 5.50, and 1.14 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 4.83 MB and 5.10 MB, respectively.

**4.1.8 Experimental Results: Incremental Backward Algorithm**

Table 4.3 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. The column head “IncBack” refers to the incremental backward algorithm. We choose a randomly generated policy $\psi_2$, 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

\[6\] the additional overhead for storing the graph and the set of airs is 0.58 seconds – 1.04 seconds.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Nodes</th>
<th>Edges</th>
<th>Time (NonIncBack)</th>
<th>Time (IncBack)</th>
<th>Nodes</th>
<th>Edges</th>
<th>Time (NonIncBack)</th>
<th>Time (IncBack)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add can_revoke</td>
<td>37112</td>
<td>286009</td>
<td>22.81</td>
<td>2.86</td>
<td>15566</td>
<td>87945</td>
<td>44.49</td>
<td>28.55</td>
</tr>
<tr>
<td>add can_assign</td>
<td>37112</td>
<td>287046</td>
<td>46.14</td>
<td>3.6</td>
<td>15566</td>
<td>88145</td>
<td>51</td>
<td>1.56</td>
</tr>
<tr>
<td>delete can_revoke</td>
<td>37112</td>
<td>286009</td>
<td>46.06</td>
<td>5.69</td>
<td>15566</td>
<td>87945</td>
<td>94.17</td>
<td>5.22</td>
</tr>
<tr>
<td>delete can_assign</td>
<td>37085</td>
<td>285226</td>
<td>32.56</td>
<td>2.47</td>
<td>15560</td>
<td>87774</td>
<td>43.52</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Table 4.3: Performance comparison of NonIncBack and IncBack on $\psi_2$ for goal$_3$ and goal$_4$.

are performed during analysis. The disk space for storing the graph is 5.33 MB and 1.72 for goal$_3$ and goal$_4$ respectively. The additional amount of disk space to store the airs is 1.67 MB and 3.71 MB for goal$_3$ and goal$_4$ respectively.

When a can_revoke rule is added or deleted, the graph remains the same. The execution time of the first stage of IncBack refers to the time for loading the graph constructed in the previous analysis. Table 4.3 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 sec and 0.97 sec of additional overhead for goal$_3$ and goal$_4$ respectively. When the previous analysis result is true, IncBack does not perform the second stage and hence the execution time of the second stage is 0. 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.

### 4.2 Incremental User-role Reachability Analysis for ARBAC without Separate Administration

This section presents IncFwdWSA, an incremental forward algorithm for ARBAC policies without separate administration.

Let $I = \langle UA_0, \psi, goal \rangle$ be the user-role reachability problem instance and $G(I)$ the graph computed from $I$ by the non-incremental algorithm. The basic idea of the algorithm is similar to IncFwd2. In this section, we only present differences between IncFwdWSA and IncFwd2.
Add a can_assign rule Suppose \( \text{can Assign}(r_a, P \land \neg N, T) \) is added to the policy. Let \( I' = \langle UA_0, \psi \cup \{ can\_assign(r_a, P \land \neg N, T) \}, goal \rangle \). The differences between IncFwdWSA and IncFwd2 are given below.

- Incremental slicing considers administrative preconditions.
- \( Rel\Rules \) is defined as follows. Let \( \text{admin}(\rho) \) represent the administrative precondition of rule \( \rho \).
  1. \( \text{can_revoke}(r'_a, r) \in Rel\Rule \) if \( r \) is a mixed role in \( I' \) or is a negative role in \( UA_0 \).
  2. \( \text{can_assign}(r_a, P \land \neg N, T) \in Rel\Rule \).
  3. a can_assign rule \( \rho \in Rel\Rule \) if
     a. \( \text{target}(\rho) \in Rel_+(I') \) and there exists \( \rho' \in Rel\Rule \) such that \( \text{target}(\rho') \in \text{poscond}(\rho) \) or \( \text{target}(\rho') = \text{admin}(\rho) \), or
     b. \( \text{target}(\rho) \) is a positive role relevant to \( T \), or
     c. \( \text{target}(\rho) \) is a positive role relevant to \( r'_a \)

- Theorem 6 is extended to consider multiple users.

Theorem 9 Let \( \text{init} = \text{closure}(\text{InitRm}_I(UA_0), I) \) and \( \text{init}' = \text{closure}(\text{InitRm}_I(UA_0), I') \). One of the following holds: (1) \( \text{init} = \text{init}' \); (2) \( \text{init}' = \text{closure}(\text{init}, I') \); or (3) \( \text{init} \) is reachable from \( \text{init}' \) through a sequence of transitions: \( \text{init}' \xrightarrow{ar(u_1, r_1)} s_1 \ldots \xrightarrow{ar(u_n, r_n)} s_n \xrightarrow{ua(u_{n+1}, r_{n+1})} s_{n+1} \ldots s_{m-1} \xrightarrow{ua(u_m, r_m)} \text{closure}(\text{init} \cup s_{m-1}, I') \) where \( \{ r_1, \ldots, r_n \} \) are revocable mixed roles in \( \text{init}' \setminus \text{init} \) and \( \{ r_{n+1}, \ldots, r_m \} \) are roles in \( \text{init} \setminus \text{init}' \) that are turned from both positive and non-negative to mixed.

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

- We perform incremental slicing from \( r_a \) similar to adding \( \text{can_assign} \).
• RelRules is defined as follows. Let negpre(\(\rho\)) be the set of negative preconditions of can_assign rule \(\rho\). A rule is in RelRule if:
   
   (1) can_revoke(\(r_a, T\)) \(\in\) RelRule.
   
   (2) can_revoke(\(r'_a, r\)) \(\in\) RelRule if \(r\) is a mixed role in \(I'\) or is a negative role in \(UA_0\).
   
   (3) a can_assign rule \(\rho\) \(\in\) RelRule if
      
      (a) target(\(\rho\)) \(\in\) Rel\(_{+}\)(\(I'\)) and \(T\) \(\in\) negcond(\(\rho\)) or
      
      (b) target(\(\rho\)) \(\in\) Rel\(_{+}\)(\(I'\)) and there exists \(\rho'\) \(\in\) RelRules 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 similar to adding can_assign.

Delete a can_assign rule Suppose can_assign(\(r_a, P \land \neg N, T\)) is deleted from the policy. Let \(I' = (UA_0, \psi \setminus \{can\_assign(r_a, P \land \neg N, T)\}, \text{goal})\). We summarize the differences between IncFwd2 and IncFwdWSA below.

• Theorem 8 is extended to consider multiple users.

**Theorem 10** Let init = closure(InitRm\(_I\)(UA\(_0\), I), init' = closure(InitRm\(_I\)(UA\(_0\), I), and Invalid = \{\(r\)|(u, r) \(\in\) (init \(\setminus\) init') and \(r\) \(\in\) (\(A_T\) \(\cup\) \{\(T\}\) for some \(u\}\). One of the following holds: (1) init' = init; (2) init' = \{(u, r)|(u, r) \(\in\) init and \(r\) \(\notin\) (RevtoIrr \(\cup\) Invalid \(\cup\) \{\(S\) \(\in\) MixtoNonpos|\(S\) is revocable\}); or (3) \(G(I)\) contains the following sequence of transitions: init \(\xrightarrow{ua(u_1, r_1)} s_1 \ldots s_{n-1} \xrightarrow{ua(u_n, r_n)} \text{init'} \cup \{(u, r)\} |(u, r) \(\in\) s_{n-1} and \(r\) \(\in\) (RevtoIrr \(\cup\) Invalid \(\cup\) MixtoNonpos)\} where \(\{r_1, \ldots, r_n\}\) are both positive and non-negative roles in init' \(\setminus\) init.

• We remove invalidated transitions from multiple users.
Table 4.4: Performance comparison of NonIncWSA and Inc for goal1 and goal2

<table>
<thead>
<tr>
<th>Operation</th>
<th>goal1</th>
<th></th>
<th>goal2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Trans</td>
<td>Time (NonIncWSA)</td>
<td>States</td>
</tr>
<tr>
<td>add can_revoke</td>
<td>31256</td>
<td>171552</td>
<td>267.26</td>
<td>42961</td>
</tr>
<tr>
<td>add can_assign</td>
<td>27661</td>
<td>131365</td>
<td>314.51</td>
<td>38140</td>
</tr>
<tr>
<td>delete can_revoke</td>
<td>24107</td>
<td>127956</td>
<td>100.50</td>
<td>38751</td>
</tr>
<tr>
<td>delete can_assign</td>
<td>24401</td>
<td>129370</td>
<td>101.58</td>
<td>39366</td>
</tr>
</tbody>
</table>

Delete a can_revoke rule Suppose \( \text{can}_{\text{revoke}}(r_a, T) \) is deleted from the policy. Let \( I' = \langle UA_0, \psi \setminus \{\text{can}_{\text{revoke}}(r_a, T)\}, \text{goal} \rangle \). We summarize the differences between IncFwd2 and IncFwdWSA below.

- Role classifications may change similar to deleting \( \text{can}_{\text{assign}} \).
- The algorithm must check if transitions revoking \( T \) are derivable through other \( \text{can}_{\text{revoke}} \) rules.
- We update the graph similar to deleting \( \text{can}_{\text{assign}} \).

4.2.1 Experimental Results

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

We apply the non-incremental and incremental forward algorithms to the university policy developed in [46]. We choose two initial RBAC policy \( UA_1 \) 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 state 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 4.1 compares the execution time of the non-incremental algorithm (NonIncWSA) against the incremental algorithm (IncFwdWSA) for two the above policy and goals \( goal_1 \) and \( goal_2 \). 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 4.4 shows that, when a can_revoke rule is
When a can_revoke rule is deleted, IncFwdWSA is 244.74 and 331.28 times faster than NonIncWSA for goal\textsubscript{1} and goal\textsubscript{2}, respectively. When considering only the 8 rules that revoke mixed roles are considered, IncFwdWSA is 12.87 times faster than NonIncWSA for goal\textsubscript{1} and 1.22 times faster than NonIncWSA for goal\textsubscript{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\textsubscript{1} and goal\textsubscript{2}, respectively.

When a can_assign rule is added, IncFwdWSA is 10.39 and 5.42 times faster than NonIncWSA for goal\textsubscript{1} and goal\textsubscript{2}, respectively. When considering only 13 rules that add roles relevant to goal\textsubscript{1}, IncFwdWSA is 8.40 times faster than NonIncWSA. When considering only 13 rules that add roles relevant to goal\textsubscript{2}, Inc 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\textsubscript{1} and goal\textsubscript{2}, respectively.

When a can_assign rule is deleted, IncFwdWSA is 42.16 and 45.62 times faster than NonIncWSA for goals goal\textsubscript{1} and goal\textsubscript{2}, respectively. When considering only the 4 rules that add roles positive roles, IncFwdWSA is 5.27 times faster and 5.70 times faster for goal\textsubscript{1} and goal\textsubscript{2}, respectively.

When both goals are considered, IncFwdWSA is 44.47 times faster than NonIncWSA. The additional overhead of saving the graph is 0.47 and 1.29 seconds for goal\textsubscript{1} and goal\textsubscript{2}, respectively.
Disk space consumption  The amount of disk space used by NonIncWSA to store the transition graph is 17.39 MB for goal1, and 52.4 MB for goal2. These numbers are much larger than the amount of disk space used by NonInc (Section 4.1.6) because the size of the state in the graph generated by NonIncWSA is much larger.

4.3 Other Analysis Problems

This section presents incremental algorithms for solving role-role containment, and user-role availability, weakest precondition and dead role analysis problems.

4.3.1 User-role Availability

User-role availability problem [27; 46] asks if a user u is always a member of a role r? To solve user-role availability analysis problem for evolving ARBAC, we reduce the problem to the user-role reachability analysis problem by adding a rule can_assign(\neg r, r') to the policy and then applying algorithms in Sections 4.1.1 and 4.1.5 to check if r' is reachable. If so, the algorithm returns false; otherwise, true.

4.3.2 Role-role Containment

Role-role containment problem [28; 46] asks: ”in every state reachable from a given initial state, is every member of role r1 also a member of role r2”? To solve role-role containment analysis problem for evolving ARBAC, we reduce the problem to the user-role reachability analysis by adding rule can_assign(r1 \land \neg r2, r) to the policy and then applying algorithms in Sections 4.1.1 and 4.1.5 to check if r is reachable from the initial state. If so, the algorithm returns false; otherwise true.

4.3.3 Weakest Precondition

The weakest precondition is the minimal sets of initial roles necessary for assigning the target user to the roles in the goal [46]. The weakest preconditions can be computed by constructing the backward graph and computing the smallest initial node (i.e., nodes that are subsets of the initial RBAC policy), from which the goal is reachable. As an optimization, the algorithm processes the initial nodes in the ascending order of size; the algorithm stops as soon as an initial node, from which the goal is reachable, is found.
Let \( I = \langle \psi, \text{goal} \rangle \) be the weakest precondition analysis problem instance, \( G_b \) be the backward graph for \( I \), and \( WP(I) \) be the weakest precondition computed for \( I \). To support incremental analysis, and mark the 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.

**Add can_revoke/can_assign** Assume that \( can\_revoke(T) \) is added to \( \psi \). First, we back-chain from the edges containing role \( T \) in the negative precondition and check if there exists a backward-reachable node \( UA_T \) containing role \( T \); if not, the result remains the same. Otherwise, we compute a set \( S_L \) of initial nodes smaller than \( WP(I) \) that are backward reachable from each node \( UA_T \). The smallest node in \( S_L \) from which the goal is reachable is the weakest precondition.

Adding \( can\_assign(P \land \neg N, T) \) is handled similarly except that \( S_L \) consists of initial nodes smaller than \( |WP(I)| \) that are backward reachable from the new edges.

**Delete can_revoke/can_assign** Assume that rule \( can\_revoke(T) \) is deleted from \( \psi \). If the goal is still reachable from \( WP(I) \) after the rule is deleted, then the algorithm returns \( WP(I) \). Otherwise, we back-chain from edges containing \( T \) in the negative precondition and check if there exists a backward reachable node \( U_T \) containing \( T \). If not, the result remains the same. Otherwise, we process initial nodes marked with 1. If the goal is not reachable such nodes, then we compute the weakest precondition from nodes not processed in the previous analysis i.e. nodes not marked with 1 or 0.

Assume that rule \( can\_assign(P \land \neg N, T) \) is deleted from \( \psi \). If the goal is still reachable from \( WP(I) \) after the rule is deleted, then the algorithm returns \( WP(I) \). Otherwise, we incrementally update the graph and process the set of initial nodes marked with 1. If the goal is not reached, then we compute the weakest precondition from nodes not processed in the previous analysis.
4.3.4 Dead Role Analysis

Dead role analysis [13] computes a set of roles that cannot be assigned to any users, 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. [13] proposed the following optimizations: (1) apply the slicing transformation to eliminate roles and rules not useful for adding unassigned roles; (2) only consider users with distinct sets of roles; and (3) first consider the users with 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 [13] to store the set of dead roles. If there are no dead roles, we mark the last user considered and all users that were not considered with 0. We then handle changes to the ARBAC policy as follows.

**Add can_revoke/can_assign**  If the policy does not contain dead roles, then adding a rule does not affect the analysis result. Otherwise, we pick a user who 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 marked with 0 have been processed and the set of dead roles is not empty, then the algorithm picks a user with most roles that are positively relevant to the added rule and performs non-incremental algorithm. Repeat the above process until all roles have been assigned or all users have been considered.

**Delete can_revoke/can_assign**  Assume that can_assign\((P \land \neg N, T)\) is deleted from the policy. First, we compute a set of roles \(A\) that are reachable through \(T\) and are not in the initial state, similar to Section 4.1.1. We then select a user not marked with 1 whose graph contains the most roles in \(A\), incrementally update the graph using LazyInc, and update \(A\).
Repeat the above process until $A$ becomes empty or all users have been processed.

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 Incremental Information Flow Analysis of Role Based Access Control

Although RBAC provides a flexible mechanism to control the release of information, it does not control how the information is propagated after it is obtained. For example, if a user has permission to read from an object $O_1$ and to write to an object $O_2$, then this user can write the content of $O_1$ to $O_2$. As a consequence, a user who has permission to access $O_2$ but not $O_1$ will be able to read the content of $O_1$ through $O_2$. Formally analyzing information flows allowed by an RBAC policy helps security administrators understand the policy better and detect potential flaws in the policy. Due to their sheer size and complexity, the full implications of RBAC policies in large organizations can be difficult to understand by manual inspection alone. To address this problem, Osborn [33] presented an algorithm for constructing an information flow graph from an RBAC policy. Each node in the information flow graph is an object. There is an edge $O_1 \rightarrow O_2$ in the graph if information can flow directly from object $O_1$ to object $O_2$, i.e., if there exists a user who has the permission to read from $O_1$ and write to $O_2$. In this proposal, we optimize this algorithm to reduce the time and space consumed for constructing the information flow graph. We also support queries such as “can information flow, directly or transitively, from object $O_1$ to object $O_2$?” and provide the corresponding evidence.

Further, RBAC policies tend to evolve incrementally over time, due to changes performed through either Administrative Role-Based Access Control (ARBAC) or other Administrative models. It would be inefficient for the analysis algorithm to reconstruct the graph from scratch if changes to an RBAC policy result in small or no changes to the information flow graph. In this proposal, we present algorithms for incrementally updating the information flow graph upon changes to policies by reusing the results obtained.
from the previous analysis. All of our incremental algorithms have the same or better worst-case complexity than our non-incremental algorithm. We have also developed an algorithm for randomly generating a relation based on the university RBAC policy in [46] and used it to compare our incremental algorithms against our non-incremental algorithm. The performance results show that our incremental algorithms significantly outperform our non-incremental algorithm in terms of time. Our incremental algorithms also require only moderately larger disk space, which is smaller than the size of the policy.

5.1 Information Flow Analysis of RBAC

In this section, we first summarize the information flow analysis algorithm presented in [33], and then present optimizations to improve its performance.

**Information flow analysis algorithm in [33].** Osborn [33] proposed an algorithm for generating an information flow graph from an RBAC policy. For simplicity, the author assumes the only operations on objects are read and write; other permissions can be represented as read and write permissions. The algorithm consists of two stages. In the first stage, the algorithm generates a cyclic information flow graph from the RBAC policy. There is an edge \((R_1, O_1, P_1) \rightarrow (R_2, O_2, P_2)\) in the graph if one of the following three conditions is satisfied: (1) \(P_1 = r, P_2 = w\), a user is assigned both roles \(R_1\) and \(R_2\), \((R_1, R_2) \notin DSD\), \(R_1\) can read from \(O_1\), and \(R_2\) can write to \(O_2\); (2) \(P_1 = r, P_2 = w, R_1 = R_2\), \(R_1\) appears in UA, and \(R_1\) can read from \(O_1\) and write to \(O_2\); or (3) \(O_1 = O_2\), both \(R_1\) and \(R_2\) appear in UA, \(R_1\) has privilege \(P_1\) on \(O_1\), and \(R_2\) has privilege \(P_2\) on \(O_1\). In the second stage, the algorithm merges nodes in every cycle of the graph, which produces an acyclic graph.

**Optimizations.** We optimize the algorithm in [33] to reduce the size of the cyclic graph generated in the first stage as well as the overhead of the second stage for detecting and eliminating cycles. First, our algorithm avoids generating edges that do not capture actual flows of information, but are used in [33] to construct cycles. Such edges are of the form
Algorithm 1 Information Flow Analysis Algorithm

1: Procedure info_graph()
2:  trans_pa_rh()
3:  trans_dsd_rh()
4:  for all \((R, O_1, r), (R, O_2, w) \in PA\) such that \(O_1 \neq O_2\) do
5:    if \(R\) appears in UA then
6:      add edge\(((R, O_1) \rightarrow (R, O_2))\)
7:    end if
8:  end for
9:  for all \(U \in User\) do
10:    for all \((U, R_1), (U, R_2) \in UA\) do
11:      if \(R_1 \neq R_2\) and \((R_1, R_2) \notin DSD\) then
12:        for all \((R_1, O_1, r), (R_2, O_2, w) \in PA\) where \(O_1 \neq O_2\) do
13:          add edge\(((R_1, O_1) \rightarrow (R_2, O_2))\)
14:        end for
15:      end if
16:    end for
17:  end for
18:  for all \(O \in Obj\) do
19:    for all \((R_1, O, w), (R_2, O, r) \in PA\) where \(R_1 \neq R_2\) do
20:      if both \(R_1\) and \(R_2\) appear in UA then
21:        add edge\(((R_1, O) \rightarrow (R_2, O))\)
22:      end if
23:    end for
24:  end for
25:  Procedure trans_pa_rh()
26:  for all \(R_1 \in Role\) do
27:    junior\((R_1) = |\{R | R \succ R_1\}|\)
28:  end for
29:  \(W = \{R_1 | \text{junior}(R_1) = 0\}\)
30:  while \(W \neq \emptyset\) do
31:    remove \(R_1\) from \(W\)
32:    for all \(R_2 \succ R_1\) do
33:      for all \((R_1, O, p) \in PA\) do
34:        add_perm\((R_2, O, p)\)
35:      end for
36:      if \((-\text{junior}(R_2) = 0)\) then
37:        \(W = W \cup \{R_2\}\)
38:      end if
39:    end for
40:  end while
41:  Procedure trans_dsd_rh()
42:  \(W = \{(R_1, R_2) | (R_1, R_2) \in DSD\}\)
43:  while \(W \neq \emptyset\) do
44:    remove \((W_1, W_2)\) from \(W\)
45:    for all \((SW_1 \succeq W_1)\) do
46:      for all \((SW_2 \succeq W_2)\) do
47:        if \((SW_1, SW_2)\) is not already in DSD then
48:          add \((SW_1, SW_2)\) to DSD
49:          add \((SW_1, SW_2)\) to \(W\)
50:        end if
51:      end for
52:    end for
53:  end while
((R_1, O, r) \rightarrow (R_2, O, r), (R_1, O, w) \rightarrow (R_2, O, w), and (R_1, O, r) \rightarrow (R_2, O, w). Second, the r/w privilege is removed from the graph as it can be inferred by examining the objects in the edge: given an edge \((R_1, O_1) \rightarrow (R_2, O_2)\), if \(O_1 = O_2\), the edge represents \((R_1, O_1, w) \rightarrow (R_2, O_2, r)\); otherwise, the edge represents \((R_1, O_1, r) \rightarrow (R_2, O_2, w)\).

Such a simplification reduces the size of the cyclic graph. An acyclic graph can be generated by removing roles from the nodes of the graph and removing self-edges, and then merging nodes in each cycle to one node.

Our algorithm for constructing the cyclic information flow graph is given in Algorithm 1. Procedure \texttt{trans\_pa\_rh} translates the hierarchical permission-role relations into non-hierarchical relations: every role inherits all permissions of its junior roles. The translation process is performed bottom-up (i.e., from junior roles to senior roles) and guarantees that each role in the role hierarchy will be processed at most once. Procedure \texttt{trans\_dsd\_rh} converts hierarchical DSD constraints to non-hierarchical DSD constraints. Procedure \texttt{add\_perm}(perm) checks if a permission \(perm\) is in \(PA\), and if not, adds \(perm\) to \(PA\). Procedure \texttt{add\_edge}(e) checks if an edge \(e\) already exists in the information flow graph, and if not, adds \(e\) to the graph. Due to space constraints, we omit the details of procedures \texttt{add\_edge} and \texttt{add\_perm}.

In our implementation, we use adjacency-list to store the information flow graph. The worst-case complexity of Algorithm 1 is \(O(|Role|^5 + |User||Role|^3|Obj|^3)\) where \(User\), \(Role\), and \(Obj\) are sets of users, roles, and objects, respectively.

**Example 14** Consider the following RBAC policy \(P\).

\[
\begin{align*}
UA &= \{ (U_1, R_3), (U_2, R_3), (U_3, R_2), (U_4, R_1), (U_5, R_1) \} \\
PA &= \{ (R_1, O_1, r), (R_1, O_2, w), (R_2, O_1, r), (R_2, O_2, r), (R_3, O_3, r), (R_3, O_3, w) \} \\
DSD &= \emptyset, RH &= \{ R_3 \succ R_1, R_3 \succ R_2 \}
\end{align*}
\]

Figure 5.1(a) gives the cyclic information flow graph generated from policy \(P\) using the algorithm in [33], which contains 9 nodes and 25 edges. The information flow graph
Figure 5.1: (a) The information flow graph generated using the algorithm in [33]. (b) The information flow graph generated using Algorithm 1. (c) The information flow graph containing only objects.

Generated from Algorithm 1 is given in Figure 5.1(b), which contains 6 nodes and 8 edges. By removing roles from nodes in Figure 5.1(b) and deleting self edges, we obtain a graph containing only objects (Figure 5.1(c)). The acyclic graph generated using the algorithm in [33] contains one edge \{O_2, O_3\} → \{O_1\}, which can be generated from Figure 5.1(c) by merging nodes containing objects O_2 and O_3.

Discussion. The algorithm in [33] considers the information flows resulting from every individual user in the RBAC policy. We observe that users assigned the same set of roles result in identical information flows. Thus, it is sufficient to consider only users with distinct sets of roles, which is usually significantly smaller than the total number of users. Users assigned the same set of roles are said to be in the same equivalence class. For example, in a university RBAC policy we developed, there are 15988 users and 33 equivalence classes. However, computing the set of equivalence classes has the worst-case complexity of \(O(|User|^2|Role|^2)\), which increases the worst-case complexity of Algorithm 1. The performance results show that using equivalence classes does not improve the performance.
of Algorithm 1 on the university RBAC policy (Section 5.4).

5.2 Incremental Information Flow Analysis

In this section, we present algorithms for incrementally updating the information flow graph in response to changes to the RBAC policy. Our incremental algorithms reuse the results of the previous analysis, including the information flow graph, the non-hierarchical permission-role relations, and the non-hierarchical DSD constraints. All of our incremental algorithms have the same or better complexity results than Algorithm 1 (the non-incremental algorithm).

Revising Algorithm 1 to support incremental analysis. To support incremental analysis, we associate each permission-role relation, DSD constraint, and edge in the information flow graph with a counter. The initial values of all counters are 0. Every time a permission-role relation, a DSD constraint, or an edge is added, the corresponding counter is increased by 1. Note that the counter of a permission-role relation may be smaller than the total number of derivations of the relation. For example, the counter of a permission-role relation \((R_1, O, p)\) is defined as: \(\text{counter}(PA(R_1, O, p)) = ((R_1, O, p) \in PA ? 1 : 0) + \{|R| R_1 \trianglerighteq R \land \text{counter}(PA(R, O, p)) > 0\}|.\) Assume that \(RH = \{R_3 \triangleright R_1, R_3 \triangleright R_2, R_2 \triangleright R_4\}\) and \(PA = \{(R_1, O, p), (R_2, O, p), (R_3, O, p), (R_4, O, p)\}.\) Then \(\text{counter}(PA(R_4, O, p)) = \text{counter}(PA(R_1, O, p)) = 1, \text{counter}(PA(R_2, O, p)) = 2,\) and \(\text{counter}(PA(R_3, O, p)) = 3,\) but the number of derivations of \((R_3, O, p)\) is 4. The advantage of such a representation is that, if \((R_4, O, p)\) is deleted from the policy, we simply decrement the counter of \((R_2, O, p)\) by 1 and the counters of other relations remain the same. However, if the counter records the total number of derivations, we will also need to decrement the counter of \((R_3, O, p)\). Similarly, the counter of an edge of the information flow graph is also smaller than the total number of derivations of this edge.

Incremental analysis algorithms. Due to space constraints, we provide the pseudo-codes
for only some of the incremental algorithms.

**Algorithm 2** Incremental Algorithm: add\_ua(U, R)

1: procedure add\_ua(U, R)
2:   for all (U, R1) ∈ UA where R1 ≠ R and (R1, R) ∉ DSD do
3:     for all (R, O1, r), (R1, O2, w) ∈ PA where O1 ≠ O2 do
4:       add\_edge((R, O1) → (R1, O2))
5:     end for
6:   for all (R, O1, w), (R1, O2, r) ∈ PA where O1 ≠ O2 do
7:     add\_edge((R1, O2) → (R, O1))
8:   end for
9: end for
10: if R has not been assigned to other users then
11:   for all (R, O1, r), (R, O2, w) ∈ PA where O1 ≠ O2 do
12:     add\_edge((R, O1) → (R, O2))
13: end for
14: for all O ∈ Obj do
15:   for all (R, O, r), (R1, O, w) ∈ PA where R1 ≠ R do
16:     if there exists (U1, R1) ∈ UA then
17:       add\_edge((R1, O) → (R, O))
18:     end if
19:   end for
20:   for all (R, O, w), (R1, O, r) ∈ PA where R1 ≠ R do
21:     if there exists (U1, R1) ∈ UA then
22:       add\_edge((R, O) → (R1, O))
23:     end if
24:   end for
25: end for
26: end if

Add a user-role relation. When a new user-role relation (U, R) ∈ UA is added to the RBAC policy, the incremental algorithm adds information flows resulting from R and all other roles assigned to U, to the graph. The corresponding algorithm is given in Algorithm 2 and its worst-case complexity is $O(|User|\cdot|Role|^2\cdot|Obj|^3)$.

Delete a user-role relation. When a user role relation (U, R) ∈ UA is removed from the RBAC policy, the incremental algorithm checks if R has been assigned to any user other than U. If so, the counters of all edges of the form (R, O) → (R1, O1) and (R1, O1) → (R, O), where $R ≠ R_1$ and $O ≠ O_1$, are decreased by 1 and all edges whose counters reach 0 are deleted. Otherwise, the algorithm deletes all edges containing role R. The worst-case complexity of this algorithm is $O(|User|\cdot|Role| + |Role|^2\cdot|Obj|^2)$. 72
Algorithm 3 Incremental Algorithm: add_\_pa(R, O, r)

1: procedure add_\_pa(R, O, r)
2: if (R, O, r) \in PA then
3: \hspace{0.5em} counter(PA(R, O, r))++
4: else
5: \hspace{0.5em} add (R, O, r) to PA
6: \hspace{0.5em} counter(PA(R, O, r)) = 1
7: \hspace{0.5em} for all (U, R), (U, R_1) \in UA where R \neq R_1 do
8: \hspace{1.5em} if (R_1, R) \notin DSD then
9: \hspace{2.5em} for all (R_1, O_1, w) \in PA, O_1 \neq O do
10: \hspace{3.5em} add_edge((R, O) \rightarrow (R_1, O_1))
11: \hspace{2.5em} end for
12: \hspace{1.5em} end if
13: \hspace{0.5em} end for
14: if R appears in UA then
15: \hspace{0.5em} for all (R, O_1, w) \in PA where O_1 \neq O do
16: \hspace{1.5em} add_edge((R, O) \rightarrow (R, O_1))
17: \hspace{0.5em} end for
18: \hspace{0.5em} for all (R_1, O, w) \in PA where R_1 \neq R do
19: \hspace{1.5em} if R_1 appears in UA then
20: \hspace{2.5em} add_edge((R_1, O) \rightarrow (R, O))
21: \hspace{1.5em} end if
22: \hspace{0.5em} end for
23: end if
24: S = \{role|role \succ R\}
25: \hspace{0.5em} for all role \in S do
26: \hspace{1.5em} add_\_pa(role, O, r)
27: \hspace{0.5em} end for
28: end if

Add a permission-role relation. When a permission-role relation (R, O, r) \in PA is added to an RBAC policy, all roles that are senior to R inherit permission (O, r) from R. The newly generated permission-role relations may result in new edges in the information flow graph. The corresponding algorithm is given in Algorithm 3 and its worst-case complexity is \(O(|User||Role|^3|Obj|^2)\). Adding (R, O, w) \in PA is handled similarly.

Delete a permission-role relation. When a permission-role relation (R, O, r) \in PA is deleted from an RBAC policy, the counter of (R, O, r) \in PA is decreased by 1. If the counter does not become 0, the algorithm terminates. Otherwise, all edges of the form (R, O) \rightarrow (R_1, O_1) where O \neq O_1 and (R_1, O) \rightarrow (R, O) are deleted. The algorithm repeats the above procedure for all roles senior to R until no more permission-role relations can be deleted. Deletion of permission-role relations (R, O, w) \in PA is handled similarly.
The worst-case complexity of this algorithm is $O(|Role|^2|Obj|)$.

**Add a DSD constraint.** The DSD constraint $(R_1, R_2) \in DSD$ specifies that no user is allowed to activate roles $R_1$ and $R_2$ simultaneously in one session. When a new DSD constraint $(R_1, R_2)$ is added, we also add $(SR_1, SR_2)$, where $SR_1 \succeq R_1$ and $SR_2 \succeq R_2$, to the DSD constraints. As a result, all edges of the form $(SR_1, O_1) \rightarrow (SR_2, O_2)$ and $(SR_2, O_1) \rightarrow (SR_1, O_2)$ are deleted from the information flow graph.

The corresponding algorithm is given in Algorithm 4 and its worst case complexity is $O(|Role|^5 + |Role|^3|Obj|^2)$.

Note that adding a new DSD constraint may result in conflicts in an RBAC policy. For example, assume that $R_3 \succ R_1$ and $R_3 \succ R_2$. Adding $(R_1, R_2) \in DSD$ to $DSD$ results in a new DSD constraint $(R_3, R_3) \in DSD$, which can never be satisfied. Such a conflict can be easily detected before applying the incremental algorithm.

---

**Algorithm 4 Incremental Algorithm: add_dsd($R_1, R_2$)**

```
1: procedure add_dsd($R_1, R_2$)
2: if $(R_1, R_2) \notin DSD$ then
3: add $(R_1, R_2)$ to $DSD$
4: counter(DSD($R_1, R_2$)) = 1
5: else
6: counter(DSD($R_1, R_2$))+
7: end if
8: $S = W = \{(R_1, R_2)\}$
9: while $W \neq \emptyset$ do
10: remove $(W_1, W_2)$ from $W$
11: for all $SW_1 \succeq W_1, SW_2 \succeq W_2$ where $(SW_1, SW_2) \neq (R_1, R_2)$ do
12: if $(SW_1, SW_2) \in DSD$ then
13: counter(DSD($SW_1, SW_2$))++
14: else
15: add $(SW_1, SW_2)$ to $DSD$
16: counter(DSD($SW_1, SW_2$)) = 1
17: $W = W \cup \{(SW_1, SW_2)\}$
18: $S = S \cup \{(SW_1, SW_2)\}$
19: end if
20: end for
21: end while
22: for all $(SR_1, SR_2) \in S$ do
23: delete all edges of the form $(SR_1, O_1) \rightarrow (SR_2, O_2)$ and $(SR_2, O_1) \rightarrow (SR_1, O_1)$ where $O_1 \neq O_2$
24: end for
```
Delete a DSD constraint. Deleting a constraint \((R_1, R_2) \in DSD\) decrements the counter associated with the constraint as well as all counters derived from it through the role hierarchy. Any constraint whose counter reaches 0 is removed. Next, for each constraint \((SR_1, SR_2) \in DSD\) that is removed, the algorithm adds the information flows resulting from users who are assigned both roles \(SR_1\) and \(SR_2\) to the graph. The worst-case complexity of this algorithm is \(O(|Role|^5 + |User||Role|^3|Obj|^3)\).

Add a role hierarchy relation. Adding a role hierarchy relation may result in new permission-role relations and new DSD constraints. When a role hierarchy relation \(R_2 \succ R_1\) is added, the incremental algorithm adds permission-role relations and DSD constraints that can be derived through \(R_2 \succ R_1\), and deletes all edges that are invalidated by the new DSD constraints. The algorithm is given in Algorithm 5 and its worst-case complexity is \(O(|User||Role|^3|Obj|^3)\).

Delete a role hierarchy relation. Deleting a role hierarchy relation may result in deletion of permission-role relations and DSD constraints. When \(R_2 \succ R_1\) is deleted, permissions of \(R_2\) and permissions of its senior roles that are inherited from \(R_1\) are deleted. The DSD constraints that are derived through \(R_1 \succ R_2\) are also deleted. The worst-case complexity of this algorithm is \(O(|User||Role|^3|Obj|^3)\).

Example 15 Consider the RBAC policy \(P\) given in Example 14. Figure 5.1(b) gives the corresponding cyclic information flow graph. Each permission is associated with a counter: \(\text{counter}(PA(R_1, O_1, r)) = \text{counter}(PA(R_1, O_2, w)) = \text{counter}(PA(R_2, O_1, r)) = \text{counter}(PA(R_2, O_2, r)) = \text{counter}(PA(R_3, O_2, r)) = \text{counter}(PA(R_3, O_2, w)) = \text{counter}(PA(R_3, O_3, r)) = \text{counter}(PA(R_3, O_3, w)) = 1\), and \(\text{counter}(PA(R_3, O_1, r)) = 2\). When a permission-role relation \((R_3, O_1, r)\) is added to \(P\), \(\text{counter}(PA(R_3, O_1, r))\) is increased by 1 (i.e., = 3) and the information flow graph remains the same. When a user-role relation \((U_1, R_1)\) is added to \(P\), edges \((R_1, O_1, r) \rightarrow (R_3, O_2, w), (R_1, O_1, r) \rightarrow (R_3, O_3, w), (R_3, O_1, r) \rightarrow (R_1, O_2, w), and
Algorithm 5 Incremental Algorithm: Add $R_2 \succ R_1$

1: procedure add_rh($R_2, R_1$)
2: add $R_2 \succ R_1$ to RH
3: $S = \emptyset$
4: for all $(R_1, R_3) \in DSD$ do
5: \hspace{1em} $W = \{R_2\}$
6: \hspace{1em} while $W \neq \emptyset$ do
7: \hspace{2em} Remove $W_1$ from $W$
8: \hspace{2em} if $(W_1, R_3) \notin DSD$ then
9: \hspace{3em} add $(W_1, R_3)$ to $DSD$
10: \hspace{3em} $counter(DSD(W_1, R_3)) = counter(DSD(R_1, R_3))$
11: \hspace{2em} $W = W \cup \{R | R \succ W_1\}$
12: \hspace{2em} $S = S \cup \{(W_1, R_3)\}$
13: \hspace{2em} else
14: \hspace{3em} $counter(DSD(R_2, R_3)) += counter(DSD(R_1, R_3))$
15: \hspace{2em} end if
16: \hspace{2em} end while
17: end for
18: for all $(R_4, R_5) \in S$ do
19: \hspace{1em} delete all edges of the form $(R_4, O_1) \rightarrow (R_5, O_2)$ and $(R_5, O_2) \rightarrow (R_4, O_1)$ where $O_1 \neq O_2$
20: end for
21: for all $(R_1, O, p) \in PA$ do
22: \hspace{1em} add_pa($R_2, O, p$)
23: end for

$(R_3, O_3, r) \rightarrow (R_1, O_2, w)$ are added to the graph. When a relation $R_3 \succ R_1$ is deleted from the role hierarchy of $P$, the counter of $(R_3, O_1, r)$ becomes 1 and the counter of $(R_3, O_2, w)$ becomes 0. As a consequence, the edge $(R_3, O_2) \rightarrow (R_2, O_2)$ is deleted from the graph.

5.3 Extensions

This section extends the incremental analysis algorithms to support parameterized RBAC, to query information flow properties, and to handle a sequence of additions and deletions of RBAC relations.

5.3.1 Information Flow Analysis of Parameterized Role Based Access Control

PRBAC (Parameterized RBAC) [45] extends the classical RBAC model with parameters to allow greater flexibility, scalability, and expressive power. Each role in PRBAC has the form $r(p_1 = x_1, \ldots, p_n = x_n)$ where $p_i$ is a distinct parameter name and $x_i$ is a data value or a variable symbolically representing data values. We use identifiers starting with
lower-case letters to represent data values, and identifiers starting with upper-case letters to represent variables. A role is said to be concrete if it does not contain variables. For example, a role for students who are taking CS101 can be specified as a concrete role $\text{Student}(\text{dept} = \text{cs}, \text{cid} = 101)$ where $\text{dept}$ represents the department ID and $\text{cid}$ represents the course number. Objects in PRBAC policies are parameterized in a similar manner.

Let $PUA$ be a set of parameterized user-role relations and $PPA$ be a set of parameterized permission-role relations. A user-role relation $(U, R) \in PUA$ in PRBAC specifies that user $U$ is a member of a concrete role $R$. The permission-role relation $(R, O, P) \in PPA$ specifies that role $R$ has permission $P$ on an object $O$; $R$ may or may not be a concrete role. Similar to RBAC, the DSD constraint in PRBAC specifies roles that cannot be invoked simultaneously in one session and the role hierarchy defines a partial order relation among roles. The information flow graph can be constructed from a PRBAC policy as follows: there is an edge $(R_1, O_1) \rightarrow (R_2, O_2)$ in the graph if (1) there exist a user $U$ who is assigned both roles $R_1$ and $R_2$, two permission-role relations $(R_1', O_1', r)$ and $(R_2', O_2', w)$, and two substitutions $\sigma_1$ and $\sigma_2$ such that $R_1 = R_1' \sigma_1$, $O_1 = O_1' \sigma_1$, $R_2 = R_2' \sigma_2$, and $O_2 = O_2' \sigma_2$; or (2) there exist two permission-role relations $(R_1', O_1', w)$ $(R_2', O_1', r)$ and two substitutions $\sigma_1$ and $\sigma_2$ such that $R_1 = R_1' \sigma_1$, $O_1 = O_1' \sigma_1 = O_2' \sigma_2$, and $R_2 = R_2' \sigma_2$.

**Example 16** Consider the following PRBAC policy:

$PUA = \{(u_1, \text{Instructor}(\text{dept} = \text{cs}, \text{cid} = 101)), (u_1, \text{Chair}(\text{dept} = \text{cs}))\}$

$PPA = \{(\text{Chair}(\text{dept} = D), \text{DeptReport}(\text{dept} = D), w), (\text{Faculty}(\text{dept} = D), \text{DeptReport}(\text{dept} = D), r), (\text{Instructor}(\text{dept} = D, \text{cid} = C), \text{GradeBook}(\text{dept} = D, \text{cid} = C), r/w)\}$

role hierarchy = \{Chair(\text{dept} = D) \succeq Faculty(\text{dept} = D)\}$

The information flow graph generated for this example is given in Figure 5.2.

### 5.3.2 Supporting Queries

We have developed algorithms to support the following queries about information flow properties: (1) Can information flow, directly or transitively, from an object $O_1$ to an object
Figure 5.2: Information flow graph generated from the PRBAC policy in Example 16.

O_2? (2) From which objects the information can flow, directly or transitively, to an object O? Both queries are answered by performing depth-first search on the information flow graph generated. Figure 5.3 gives the screenshot for answering the query “can information flow from GradeBook to DeptReport?” The answer to this query is yes. The right window provides diagnostic information. When clicking the edge between GradeBook and DeptReport, the RBAC rules that result in this information flow are highlighted: information can directly flow from GradeBook to DeptReport because the role DeptChair can read from GradeBook (inherited from the role Faculty) and write to DeptReport.

5.3.3 Adding/Deleting a Sequence of Relations

If the administrators make multiple changes to the policy at one time, it would be sufficient to execute the incremental algorithm for every change in the order in which it was provided. Such an approach, although simple, may not always be optimal. We propose to reduce the number of changes to the information flow graph by controlling the order in which changes are processed: changes to DSD constraints are processed first, followed by changes to role-hierarchy relations. Changes to user-role and permission-role relations can be done in any order. Changes to DSD constraints are processed first because they may invalidate changes to the information flow graph caused by changes to other relations. For example, adding a user-role relation or a permission-role relation and then a DSD constraint may cause the newly added edges to be deleted. Changes to role-hierarchy relations are processed next.
because changes to role-hierarchy relations result in changes to $DSD$ constraints.

5.4 Performance Results

We have developed a university RBAC policy based on the student/faculty statistics from Binghamton University. This section compares the performance of the non-incremental and incremental algorithms by executing them over a set of randomly generated RBAC relations for the university RBAC policy. All reported results were obtained on a 2.80GHz Pentium(R) D machine with 1GB RAM running Fedora Linux.

The university RBAC policy contains 33 roles, 15988 users, and 27 objects. Each data point reported is an average over 16 randomly generated RBAC relations. Due to space constraints, we only describe how we randomly generate a user-role relation to add to the university RBAC policy; other relations are generated similarly. To generate a user-role relation, we first randomly choose a $can_assign$ rule from the university ARBAC policy developed in [46], which specifies the authority to assign users to roles. For example, $can_assign(\text{DeptChair}, \text{Grad} \land \neg RA, TA)$ specifies that the role $\text{DeptChair}$ has authority to assign a user who is a member of $\text{Grad}$ but not a member of $RA$ to the role $TA$; $TA$ is called the postcondition of this rule. Next, we randomly choose a user $u$ who has not been
assigned the postcondition \( r \) of the rule chosen and add \((u, r)\) to \( UA \).

Table 5.1 gives the number of nodes and edges generated, and the execution time of the following four algorithms on the university RBAC policy: the non-incremental algorithm without equivalence classes (NonInc), the non-incremental algorithm with equivalence classes (NonIncEQ), the incremental algorithm without equivalence classes (Inc), and the incremental algorithm with equivalence classes (IncEQ). Column “Operation” specifies the operations on the policy.

The performance results show that the incremental algorithms are 100-400 times faster than the non-incremental algorithm. A total of 33 equivalence classes are constructed from the university policy, which is significantly smaller than the number of users in the policy. We observe that using equivalence classes does not improve the performance of the non-incremental algorithm due to the overhead introduced by constructing the equivalence classes from the set of users, as described in Section 5.1. The incremental algorithms with equivalence classes reuse the equivalence classes constructed from the previous analysis and hence avoid the above overhead. As a result, the incremental algorithms with equivalence classes are around 3 times faster than the incremental algorithms without equivalence classes. The additional amount of disk space used to store information between analysis runs is 222.9KB with equivalence classes and is 139.9KB without equivalence classes, which is less than the size of the university RBAC policy (421.6KB).

Table 5.1: Performance comparison of the non-incremental algorithm and the incremental algorithms

<table>
<thead>
<tr>
<th>Operation</th>
<th>States</th>
<th>Trans</th>
<th>Time (Sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>NonInc</td>
</tr>
<tr>
<td>add UA</td>
<td>142</td>
<td>3069</td>
<td>18.42</td>
</tr>
<tr>
<td>delete UA</td>
<td>142</td>
<td>3044</td>
<td>18.85</td>
</tr>
<tr>
<td>add PA</td>
<td>143</td>
<td>3067</td>
<td>18.52</td>
</tr>
<tr>
<td>delete PA</td>
<td>141</td>
<td>2918</td>
<td>18.77</td>
</tr>
<tr>
<td>add RH</td>
<td>146</td>
<td>3205</td>
<td>18.60</td>
</tr>
<tr>
<td>delete RH</td>
<td>121</td>
<td>2263</td>
<td>18.82</td>
</tr>
<tr>
<td>add DSD</td>
<td>142</td>
<td>2950</td>
<td>18.44</td>
</tr>
<tr>
<td>delete DSD</td>
<td>142</td>
<td>3044</td>
<td>18.85</td>
</tr>
</tbody>
</table>
6 RBAC-PAT: A Policy Analysis Tool for Role Based Access Control

We developed RBAC-PAT, a tool for analyzing RBAC and ARBAC policies, which supports analysis of various properties including reachability, availability, containment, weakest precondition, dead roles, and information flows. RBAC-PAT is based on the algorithms presented in Chapters 3, 5, and [46]. The architecture of RBAC-PAT is shown in Figure 6.1. Below, we describe its main components.

6.1 Hierarchy Converter

This component converts hierarchical policies into non-hierarchical policies for analysis [39].

6.2 Policy Analysis Engine

RBAC-PAT implements user-role reachability, user-role availability, role-role containment, weakest precondition, dead role analysis, and other algorithms discussed in Chapters 3, 4, 5.

Figure 6.1: System architecture.
It also computes policy statistics, including the numbers of mixed roles and irrevocable roles, and checks whether separate administration holds. RBAC-PAT uses this information to try to choose the most appropriate analysis algorithms for a given analysis problem. In cases where separate administration restriction is satisfied and it is unclear whether the forward or the backward algorithm will be faster, RBAC-PAT prompts the user to choose between these algorithms.

**Evidence generation.** RBAC-PAT provides evidence that shows why a property holds or is violated. For example, if the answer to a reachability analysis query is yes, RBAC-PAT provides a sequence of administrative actions that leads to the specified role assignment, and highlights the corresponding ARBAC rules in the policy.

### 6.3 Case Studies

We applied RBAC-PAT to policies for a university and a health care facility discussed in Chapter 3. Here are some sample properties for the university policy: (1) **User-role reachability:** can a user initially in role *DeptChair* and a user initially in role *Undergrad* together assign the latter user to *HonorsStudent*? (2) **Weakest Precondition:** what are the weakest preconditions for an administrator initially in *DeptChair* to assign a user to *HonorsStudent*? (3) **Role-role containment:** is *TA* contained in *Grad*? (4) **Information flow query:** can information flow from *GradeBook* to *DeptReport*? RBAC-PAT terminates in at most 0.19 second for all queries we tried. RBAC-PAT also helped uncover some flaws in the original university policy, for example, a place where we accidentally used *Student* instead of *Undergrad* and places where we forgot to take role hierarchy into account, e.g., places where we forgot that *Provost* inherits from *Staff*. Further, in order to validate our FPT results and explore the practical performance of the algorithms, we applied RBAC-PAT to reachability analysis of hundreds of randomly generated policies [46]. For the policies containing 13 reachable mixed roles and 32 roles, RBAC-PAT generates at most 232320 states and 2900920 transitions, and terminates in 8.6 hours. For the policies containing 5 reachable mixed roles and 500 roles, RBAC-PAT generates at most 510 states and 2550
transitions, and terminates in 155 minutes.
7 Related Work

Policy Analysis. We classify related work on security policy analysis into three categories.

The first, and largest, category is analysis (including enforcement) of a fixed security policy. Some representative papers in this category include [22; 2; 18; 24; 17; 20]. Work in this category is less closely related to our work, so we do not discuss it further.

The second category is analysis of a single change to a fixed policy or, similarly, analysis of the differences between two fixed policies. Jha and Reps present algorithms to analyze the effects of a specified change to a SPKI/SDSI policy [24]. Fisler et al. [10] give algorithms to compute the semantic difference of two XACML policies and check properties of the difference.

Work in the first two categories differs significantly from our work (and other work in the third category) by not considering the effect of sequences of changes to the policy.

The third category is analysis that considers sequences of changes to a policy; the allowed changes are determined by parts of the policy that we call “administrative policy”. Harrison, Ruzzo, and Ullman [19] present an access control model based on access matrices, which can express administrative policy, and show that the safety analysis problem is undecidable for that model. 7 Following this, a number of access control models were designed in which safety analysis is more tractable, e.g., [30; 36]. While those models were designed mainly with tractability in mind, we aim to provide more practical results, by starting with more a realistic model, based on ARBAC97 [37], and identifying properties of typical policies that can be exploited for efficient analysis. Our framework allows features not considered in those papers, such as negative preconditions.

7That result does not apply to ARBAC policy analysis, because the HRU model allows creation of subjects and objects, while ARBAC does not allow creation of users, roles, or permissions.
Finally, we focus on prior work on analysis of ARBAC policies.

Schaad and Moffett [41] use the Alloy analyzer [21] to check separation of duty properties for ARBAC97. They do not consider preconditions for any operations; this greatly simplifies the analysis problem. Since they leave the analysis to the Alloy analyzer, they do not present analysis algorithms or complexity results.

Li and Tripunitara [28] give algorithms and complexity results for various analysis problems—primarily safety, availability, and containment—for two restricted versions of ARBAC97, called AATU and AAR. Their results are based on Li, Mitchell, and Winsborough’s results for analysis of trust management policies [27]. Our work goes significantly beyond their analyses for both AATU and AAR by allowing negative preconditions. This forces us to consider other (more realistic) restrictions, such as bounds on the size of preconditions, and to use fixed-parameter tractability to characterize the complexity of our algorithms. In addition, our work in Section 3.2 goes significantly beyond their analysis for AAR by dropping the separate administration restriction. Sistla and Zhou [42], like [27], consider trust management policies changing in accordance with role restrictions that indicate, for each role, whether arbitrary rules defining that role may be added, and whether they may be removed. The administrative policies we consider are finer-grained than such role restrictions.

Sasturkar et al. [39] present algorithms and complexity results for analysis of ARBAC policies subject to a variety of restrictions. Our work goes beyond theirs by providing efficient algorithms for larger and more realistic classes of policies, providing fixed-parameter tractability results to more accurately characterize the complexity of those algorithms, and giving analysis algorithms that do not rely on the separate administration restriction, which is implicitly adopted throughout their paper. Also, they do not consider containment analysis.

**Incremental Computation.** 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 [15] incrementally updated materialized views in databases by counting the number of derivations for each relation. Our approach in Chapter 4 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. [16] 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 [35]. We avoid the rederivation phase by removing only those roles from the state for which all derivations have been invalidated. Lu et al. [31] 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. [5] 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 proposal extends our conference paper [14] in several ways. (1) we add incremental algorithms for the following analysis problems: availability, 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 algorithm, and present the performance results. (3) we add proofs for Theorems 6 and 8. (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 [14] explicitly stores the states in each transition, which results in a single state being stored multiple times. (5) we have also 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 [14]. (6) we add pseudocodes for the forward algorithms for deleting \textit{can_revoke}
and \textit{can_assign} rules and add examples for some of the algorithms.

The work that is most closely related to our incremental information flow analysis of
RBAC is Gupta et al.’s work on incrementally updating materialized views specified us-
ing Datalog [15]. While the incremental algorithms without equivalence classes can be
encoded as a Datalog program, our algorithm is more efficient than directly applying their
algorithm to update information flow graph due to the following reasons: (1) their algorithm
was designed for general Datalog programs. Consequently, many operations performed in
their algorithm are not necessary for incrementally updating the information flow graph.
One of such operations involves dividing a Datalog program into its strongly connected
components; and (2) the counter in their work records the total number of derivations of
each Datalog relation, while the value of our counter may be less than the total number of
derivations. As a result, our algorithm updates counters less frequently.

Sokolsky et al. [43] used approach similar to Gupta et al.’s delete-rederive [16] in their
incremental model checking algorithm (MCI). Saha et al. [35] also apply delete-rederive
approach when facts are deleted in tabled Prolog programs. Their algorithm controls the
deletion of Prolog relations based on a data structure called \textit{support graph}. The deletion al-
gorithms in Chapter 5 have lower worst-case complexity than the two-phase delete-rederive
algorithms when applied to information flow analysis. Also, the size of the support graph
in [35] is large for large Prolog programs. Although Lu et al.’s. [31] StDel algorithm is
applicable to information flow analysis, doing so would require the algorithm to store all
derivations (proofs) whose size is usually large.

Some other incremental analysis algorithms do not rely on the use of counters. Con-
way et al.’s. [6] incremental analysis algorithms for C programs is not directly applicable
to information flow analysis of RBAC because RBAC has no control flow. Incremental
computation has also been applied to data flow analysis (e.g. [34; 47]). However, they
either compute less precise answers than their non-incremental counterparts or are only
applicable to specific types of analysis problems.

**Case Studies.** Our ARBAC policy for a university contains significantly more `can_assign` rules than the ARBAC policies presented in [37; 40; 32; 41; 25; 7; 39; 28; 26], which typically contain about 4 administrative roles and the equivalent of 4 to 7 `can_assign` rules. Some papers, such as [40; 25], sketch the general structure of RBAC and ARBAC policies of very large organizations, but only a few specific administrative roles and rules are presented in the paper (or otherwise made publicly available), and no analysis algorithms were applied to those policies. We analyzed our ARBAC policy for a university. The policy contains 11 administrative roles, 21 other roles, 28 `can_assign` rules (106 rules after the transformation to eliminate role hierarchy), etc.

---

8In some cases, a few rules are duplicated, e.g., copied with change only to the name of the department; we did not count the duplicates, since we did not include such duplicates in our university policy. Also, we did not count other kinds of rules, e.g., `can_assignp`; there are only a few of those, too.
8 Research Plan

In Chapter 3 we presented a forward algorithm for ARBAC policies without separate administration. Because the algorithm is fixed-parameter tractable W.R.T. number of users, its performance is often prohibitively slow for real-world policies e.g. in our experiments, the following query ran for more than 10 days: "Given RBAC policy \{ (u_1, President), (u_2, Undergrad), (u_3, Undergrad), (u_4, Undergrad),
(u_5, Undergrad), (u_6, Undergrad), (u_7, Undergrad), (u_8, Undergrad),
(u_9, Undergrad), (u_{10}, Undergrad) \} and university ARBAC policy in [46], can user u_5 be assigned to roles \{Grad, Undergrad\}?" Hence, we propose the following optimizations:

1. **Partial Hierarchical Role Assignment:** we propose to optimize the analysis of queries where the ARBAC policy has a "partial hierarchical" role assignment, i.e., not all \textit{can_assign} rules in the policy satisfy hierarchical role assignment. In such cases we only consider administrative actions on non-target users, that are enabled through \textit{can_assign} rules that (1) do not satisfy HRA, (2) rules that enable such rules, and (3) rules whose administrative preconditions appear in the negative preconditions of rules in (1) and (2).

2. **Hierarchical Rule Reduction:** we propose to improve the performance by not considering rules that are not useful for reaching the goal. Because a user assigned to administrative role \(r_a\) is an implicit member of all roles junior to \(r_a\), we can eliminate rule \textit{can_assign}(r'_a, P \land N, r) if \(UA_0\) contains a user assigned to role \(r_a \succeq r'_a\) and eliminating the rule will not affect the analysis result.
3. *Delayed Revocation:* we propose to reduce the size of the transition graph by delaying transitions that do not enable new transitions. Transition $s \xrightarrow{ur(u,r)} s'$ cannot enable new transitions if the policy does not contain rule $\text{can\_assign}(r_a, P \land \neg N, r_1)$ where $r \in N$, $(u_a, r_a) \in s$ for some user $u_a$, $u$ satisfies $P$, and $(u, r_1) \notin s$.

4. We propose to reduce the size of the transition graph by reducing the number of users considered in each state while still ensuring correctness. Given a state $s$, if multiple non-target users in $s$ are assigned the same set of roles, then we compute transitions for only one of such users.

5. *Parallel Reachability Analysis without Separate Administration:* we will also investigate parallel algorithms for user-role reachability analysis without separate administration and evaluate them on multi-core machines.
9 Milestones

All optimizations discussed in Chapter 8 can be developed simultaneously. Upon completing the theoretical portion of the investigation, we plan to combine all optimizations in a single implementation and evaluate the performance gains when using all optimizations in concert. The Gantt chart in Figure 9.1 provides a timeline of the project milestones.

<table>
<thead>
<tr>
<th>Task</th>
<th>Month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchical Role Assignment</td>
<td>May</td>
</tr>
<tr>
<td>Hierarchical Rule Reduction</td>
<td>June</td>
</tr>
<tr>
<td>Delayed Revocation</td>
<td>July</td>
</tr>
<tr>
<td>Equivalence Classes</td>
<td></td>
</tr>
<tr>
<td>Parallel Algorithms</td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
</tr>
<tr>
<td>Evaluation</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.1: Gantt chart of the project milestones
Bibliography


