Symbolic Reachability Analysis for Parameterized Administrative Role Based Access Control*

Scott D. Stoller[1] Ping Yang[2] Mikhail Gofman[2] C. R. Ramakrishnan[1]

ABSTRACT

Role based access control (RBAC) is a widely used access control paradigm. In large organizations, the RBAC policy is managed by multiple administrators. An administrative role based access control (ARBAC) policy specifies how each administrator may change the RBAC policy. 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.

Allowing roles and permissions to have parameters significantly enhances the scalability, flexibility, and expressiveness of ARBAC policies. This paper defines PARBAC, which extends the classic ARBAC97 model to support parameters, and presents an analysis algorithm for PARBAC. To the best of our knowledge, this is the first analysis algorithm specifically for parameterized ARBAC policies. We evaluate its efficiency by analyzing its parameterized complexity and benchmarking it on case studies and synthetic policies.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*

General Terms: Security, Verification

1 Introduction

Role based access control (RBAC) [17] is a widely used access control paradigm. In RBAC, users are assigned to roles, and permissions are granted to roles. Allowing roles and permissions to have parameters significantly enhances scalability: the policies of most large organizations can be expressed more easily and compactly using parameters. For example, consider a policy for a university. To grant different permissions to users (e.g., faculty or students) in different classes or departments, in an RBAC model without parameters, we would need to create a separate role and corresponding permission assignment rules for each course or department, leading to a large and unwieldy policy. In a parameterized RBAC model, this policy can be expressed using a few roles and permissions parameterized by the class identifier or department name. Several parameterized RBAC models have been proposed, going back at least to [11].

Administrative role based access control (ARBAC) refers to administrative policies that specify how an RBAC policy may be changed by each administrator. In ARBAC97, the first comprehensive ARBAC model [16], ARBAC policies assign users (administrators) to administrative roles, and grant permissions for administrative operations—such as assigning a user to a role—to administrative roles. This supports decentralized policy administration, which is crucial for large organizations, coalitions, etc.

Allowing administrative roles and administrative permissions to have parameters significantly enhances the scalability and practical applicability of the administrative model. For example, consider the policy that the chair of a department can assign users to committees in that department. In a parameterized ARBAC model, this can be expressed by a single rule, while ARBAC models without parameters would require separate rules for each department and committee. In this paper, we define parameterized RBAC and ARBAC models, by extending the classic ARBAC97 model [16] with parameters in a fairly straightforward way. We call these models PRBAC and PARBAC, respectively.

While flexible and expressive administrative models are needed to handle the complex policies that can arise in real organizations, they also make it more difficult to ensure that administrative policies accurately capture the author's intentions. It is often difficult to understand the effect of an administrative policy by simple inspection, largely because (without help) people may fail to see the possible effects of sequences of administrative operations by different administrators, and may fail to take into account how the administrative rules interact with role hierarchy. Policy analysis helps system designers and administrators understand policies, including administrative policies. This paper focuses on user-role reachability analysis, which answers questions of the form: given an initial PRBAC policy ("state"), a PARBAC policy, a set of administrators, a target user, and a set of roles (called the "goal"), is it possible for those adminis-

*This work was supported in part by ONR under Grant N00014-07-1-0928 and NSF under Grants CCF-0613913, CNS-0627447, CNS-0831298, and CNS-0509230.

[1]Dept. of Computer Science, Stony Brook University. Email: {stoller,cram}@cs.stonybrook.edu

[2]Dept. of Computer Science, Binghamton University. Email: {pyang,mgofman1}@binghamton.edu

SACMAT'09, June 3–5, 2009, Stresa, Italy.

trators to modify the RBAC policy so that the target user is a member of those roles? Other analysis problems including permission-role reachability, user-permission reachability, availability, role containment [14], and weakest precondition [20] can be solved in a similar manner or by reduction to user-role reachability analysis [19, 20].

This paper presents the first (to the best of our knowledge) algorithm designed for reachability analysis of parameterized ARBAC policies. We define the semantics of PARBAC policies in terms of a straightforward *concrete transition relation*. We then introduce a more complicated *symbolic transition relation* that captures the semantics compactly, efficiently, and exactly using variables and constraints. Our algorithm for user-role reachability has two stages. The first stage performs a goal-directed approximate backward search. The second stage performs an exact forward search limited to transitions identified as useful by the first stage. An optimization to the second stage exploits information from the first stage to reduce the number of explored interleavings of transitions.

Why are new algorithms needed to solve this problem? If all parameters range over finite types, existing finite-state reachability algorithms for unparameterized ARBAC (e.g., [14, 19, 20, 12]) can be used, by instantiating each rule with all combinations of values of its parameters. However, this approach is practical only if the types are small. Realistic policies often involve large types (e.g., Stony Brook University has over 2000 class sections each semester and over 50 departments); symbolic analysis of such policies is much more efficient. Another disadvantage of the finite-state approach is that the analysis results are valid only for the specific types used for instantiation. With symbolic analysis, an infinite type can be used as an abstraction of a finite type, to obtain more general results. Specifically, if symbolic analysis with infinite types says that a goal (of the attackers, *i.e.*, an unsafe state) is unreachable, then that goal is unreachable when the parameters range over any finite types.

Parameters in our framework may range over infinite types, so the system is infinite-state, and reachability might be undecidable. Numerous algorithms have been proposed to verify specific classes of infinite-state systems. As discussed in Section 9, to the best of our knowledge, none is suitable for efficient analysis of PARBAC. Our algorithm is a semi-decision procedure for PARBAC reachability, guaranteed to terminate under realistic assumptions about the policy.

We also explore the parameterized complexity [7] of user-role reachability for PARBAC and give a fixed-parameter tractability result for it under realistic assumptions about the policies. The idea of parameterized complexity 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 *fixed-parameter tractable* 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.

In summary, the main contributions of this paper are (1) the definition of the symbolic transition graph, which compactly captures the semantics of PARBAC policies and provides the basis for our algorithm, (2) a two-stage symbolic algorithm for user-role reachability analysis of PARBAC that terminates under realistic assumptions about the policy, (3) a fixed-parameter tractability result for this reachability problem under realistic assumptions about the policy, and (4) experimental results demonstrating the efficiency of our algorithm compared to non-symbolic algorithms.

We chose ARBAC97 as the basis for our PARBAC model because it is relatively simple while still capturing essential features of realistic administrative policies. We know of only one other parameterized ARBAC model, UARBAC^P [13]. UARBAC^P is more sophisticated and flexible than PARBAC, but we believe the work in this paper provides a good foundation for developing practical analysis algorithms for UARBAC^P and other parameterized security policy models.

2 PRBAC and PARBAC

This section formally defines parameterized RBAC (PRBAC) and parameterized ARBAC (PARBAC). The definitions are based on a notion of *role schema*. Each role schema specifies the name of a role and the names of that role's parameters. To save space, we omit aspects of RBAC and ARBAC related to the user-permission assignment and role hierarchy. Those aspects can be extended with parameters in the same way as aspects related to the user-role assignment. For analysis purposes, hierarchical PRBAC policies can be transformed into non-hierarchical PRBAC policies using an algorithm similar to the one in [19]. PARBAC policies that control the user-permission assignment are structurally similar to PARBAC policies that control the user-role assignment and hence can be analyzed using the same techniques. Analysis of PARBAC policies that control changes to the role hierarchy requires different techniques.

2.1 Parameterized RBAC

The syntax of policies is parameterized by a set *Var* of *variables*, a set $\mathcal{R}$ of *role names*, a set $\mathcal{O}$ of *object names*, a set $\mathcal{P}$ of *parameter names*, and a set *Op* of operation names.

A role schema is a term $\rho(p_1, p_2, \ldots, p_n)$, where $n \geq 0$, $\rho \in \mathcal{R}$ is a role name, and each $p_i \in \mathcal{P}$ is a distinct parameter name. In our basic framework, each parameter can take values from an implicit universal data type that contains an infinite number of data values (constants). Introducing a type system in which each parameter in a role schema ranges over a specified infinite data type has no significant effect on our results, except to add clutter. Allowing finite types requires only a change to the algorithm for checking satisfiability of constraints, as described in Section 4. We implicitly extend our framework with a type system with finite types in some examples.

An *instance* of a role schema $\rho(p_1, p_2, \ldots, p_n)$ has the form $\rho(p_1 = x_1, p_2 = x_2, \ldots, p_n = x_n)$, where each x_i is a data value or a variable. We use identifiers starting with lower-case letters for data values, and identifiers starting with upper-case letters for variables (identifiers starting with upper-case letters are also used for role names, etc.). An instance is *concrete* if it contains no variables. We use r to denote an instance of a role schema, and r_c to denote a concrete instance.

For an instance r, let $schema(r)$ denote the schema of which r is an instance. Let $args(\rho(e_1, \ldots, e_n)) = (e_1, \ldots, e_n)$. For a set RS of role schemas, $inst(RS)$ denotes the set of all instances of RS, and $conc(RS)$ denotes the set of concrete in-

stances of *RS*. For example, in a policy for a university, the role schema *Student*(*dept*, *cid*) is used for students registered for the course numbered *cid* offered by department *dept*, and the role schema *Student*(*dept*) is used for all students of a specific department. Students taking cs101 are members of the instance *Student*($dept = cs, cid = 101$). We make parameter names explicit to allow overloading; we sometimes omit them for role names that are not overloaded.

A substitution is a mapping from variables to data values and variables. We use θ, σ to denote substitutions. A substitution θ is *ground*, denoted *ground*(θ), if it maps all variables to data values. The application of a substitution θ to an expression e is denoted $e\theta$.

Definition 1. A *parameterized RBAC (PRBAC) policy* is a tuple $\langle RS, U, UA\rangle$ where

- *RS* is a finite set of role schemas. U is a finite set of users.
- $UA \subseteq U \times conc(RS)$ is the user-role assignment. $(u, r_c) \in UA$ specifies that user u is a member of r_c.

For example, $(Alan, Student(dept = cs)) \in UA$ specifies that user *Alan* is a member of role *Student*($dept = cs$).

2.2 Parameterized ARBAC

A PARBAC policy is a tuple $\langle RS, U, URA\rangle$, where *RS* is a set of role schemas, U is a set of users, and—analogously to ARBAC97—*URA* is the user-role administration policy. The PARBAC policy defines the transition relation that describes allowed changes to the PRBAC policy.

The user-role administration policy *URA* controls changes to the user-role assignment. *URA* consists of two kinds of rules: *can_assign* and *can_revoke*. A *can_assign* rule has the form *can_assign* $(r_a, (P, N), r)$, where $r_a \in inst(RS)$ is the administrator's role, $P \subseteq inst(RS)$ is the *positive precondition*, $N \subseteq inst(RS)$ is the *negative precondition*, and $r \in inst(RS)$ is the *target*. The rule means that an administrator in role r_a can add a user to r if the user is a member of all the roles in P and is not a member of any roles in N. In examples, we usually write preconditions as logical formulas; for example, the precondition $(\{r_1, r_2\}, \{r_3\})$ would be written as $r_1 \wedge r_2 \wedge \neg r_3$. For example, *can_assign*(*Dean*($school = engg$), *Prof*($dept = cs$), *Chair*($dept = cs$)) specifies that the Dean of the Engineering School can assign a professor of the CS Department to be the Chair of that Department. The identity of the administrator performing an action is sometimes relevant, so we introduce a distinguished variable, *Self*, whose value identifies that administrator. For example, *can_assign*(*Faculty*, *Student*, *RA*($fac = Self$)) specifies that a faculty member can assign a student to be his/her RA.

A *can_revoke* rule has the form *can_revoke*(r_a, r). It means that an administrator in role r_a can remove users from role r. We follow ARBAC97 in omitting preconditions from *can_revoke* [16].

A role schema is an *administrative role schema* if it has an administrative permission, i.e., it appears in the first component of some *can_assign* or *can_revoke* rule. An *administrative role* is an instance of an administrative role schema. The *separate administration restriction* requires that administrative role schemas do not appear in the precondition or target of *can_assign* rules or the target of *can_revoke* rules. We follow ARBAC97 in adopting this restriction. Our algorithm is also applicable to many policies that satisfy a different but related restriction, described in Section 6.

3 User-Role Reachability

This section defines user-role reachability for PARBAC. For a PRBAC policy γ, let $U(\gamma)$ and $UA(\gamma)$ be the set of users and the user-role assignment in γ, respectively.

Definition 2. A *user-role reachability* query has the form: Given a user u_0, an initial PRBAC policy γ, a PARBAC user-role administration policy *URA*, a subset A of the user-role assignment $UA(\gamma)$ containing only administrative roles, and a set g of role instances, can actions by administrators in A, acting in the administrative roles to which they are assigned in A, and using the administrative permissions granted to those roles by *URA*, transform γ to another PRBAC policy γ' such that, for some substitution θ, u_0 is a member of all roles in the instantiated goal $g\theta$?

Under the separate administration restriction, the user-role reachability problem can be simplified as in [19]. This restriction implies that the transitions allowed by a PARBAC policy do not change the set of tuples containing administrative roles in the user-role assignment *UA*. Hence we can partition *UA* into administrative and non-administrative subsets, corresponding to tuples containing administrative roles and those containing non-administrative roles, respectively. Since the administrative subset does not change, we "factor it out", i.e., we do not include it in the nodes of the concrete state graph, defined below. Moreover, in ARBAC97, each user's role memberships are controlled completely independently of other users' role memberships, so we can perform user-role reachability analysis by tracking only tuples in *UA* that contain the user u_0 mentioned in the reachability query. Thus, the answer to a user-role reachability query can be expressed in terms of a graph whose vertices (states) correspond to sets of non-administrative roles that u_0 is a member of.

The *concrete transition relation* $T_c(URA, A)$ expresses the semantics of a user-role administration policy *URA*, restricted to administrative actions performed by a user u_A in administrative role r_A such that $(u_A, r_A) \in A$. $T_c(URA, A)$ contains $(s, (\varphi, \theta), s')$ iff the rule φ in *URA*, instantiated using substitution θ, allows an administrator u_A acting in role r_A with $(u_A, r_A) \in A$ to perform a role assignment or role revocation that changes the user-role assignment for a user from s to s'. When *URA* and A are clear from context, we sometimes write a triple $(s, (\varphi, \theta), s') \in T_c(URA, A)$ as $s \xrightarrow{\varphi,\theta}_c s'$.

Definition 3. The *concrete transition relation* $T_c(URA, A)$ for a user-role administration policy *URA* and a user-role assignment A containing only administrative roles is the smallest relation such that:

- $(s, (\varphi, \theta), s') \in T_c(URA, A)$ if $\varphi = can\_assign(r_a, (P, N), r)$ and $\varphi \in URA$ and θ is a ground substitution such that there exists $(u_A, r_A) \in A$ such that:
  - $r\theta \notin s$ and $s' = s \cup \{r\theta\}$,
  - $P\theta \subseteq s$, and $N\theta \cap s = \emptyset$ (the positive preconditions and negative preconditions of φ are satisfied in state s),
  - $r_a\theta = r_A$ (instantiating r_a yields the administrative role in A used to perform this role assignment), and
  - $\theta(Self) = u_A$ (θ maps the distinguished variable *Self* to the identity u_A of the administrator performing this role assignment)

- $(s, (\varphi, \theta), s') \in T_c(URA, A)$ if $\varphi = \mathit{can\_revoke}(r_a, r)$ and $\varphi \in URA$ and θ is a ground substitution such that there exists $(u_A, r_A) \in A$ such that:
  - $r\theta \in s$ (the role to be revoked is present in state s),
  - $s' = s - \{r\theta\}$,
  - $r_a\theta = r_A$, and
  - $\theta(\mathit{Self}) = u_A$

The *concrete state graph* for a user-role reachability query of the form in Definition 2 is the graph created by starting from the initial user-role assignment for the target user u_0 and using the concrete transition relation to repeatedly add new edges and nodes. The answer to a user-role reachability query is true iff there exists a substitution θ such that the concrete state graph contains a state s with $g\theta \subseteq s$. For a labeled graph, we use a triple (v, ℓ, v') to represent an edge from v to v' labeled with ℓ.

Definition 4. The *concrete state graph* for a user-role reachability query of the form in Definition 2 is the smallest labeled directed graph (V, E) with vertices V and labeled edges E such that

- $\{r \mid (u_0, r) \in UA(\gamma) \wedge \neg \mathit{admin}(r)\} \in V$, where $\mathit{admin}(r)$ is true iff r is an administrative role.
- $(s_1, \varphi, s_2) \in E$ and $s_2 \in V$ if $s_1 \in V$ and there exists a substitution θ such that $(s_1, (\varphi, \theta), s_2) \in T_c(URA, A)$.

Example 1. Consider the following PARBAC policy (for brevity, we do not show the set of users, etc.).

$RS = \{\mathit{Chair}(\mathit{dept}), \mathit{Student}(\mathit{dept}, \mathit{cid}), \mathit{TA}(\mathit{dept}, \mathit{cid})\}$
$\varphi : \mathit{can\_assign}(\mathit{Chair}(\mathit{dept} = D), \neg\mathit{Student}(\mathit{dept} = D, \mathit{cid} = \mathit{CID}),$
$\mathit{TA}(\mathit{dept} = D, \mathit{cid} = \mathit{CID}))$

The policy contains no *can_revoke* rules. Consider the query: Can the chair of CS Department assign a user u who is initially a member of role $\mathit{Student}(\mathit{dept} = cs, \mathit{cid} = 501)$ to both roles $\mathit{TA}(\mathit{dept} = cs, \mathit{cid} = 101)$ and $\mathit{TA}(\mathit{dept} = cs, \mathit{cid} = 201)$? The answer is yes. For illustrative purposes, we assume the course identifier parameter *cid* ranges over the set $\{101, 201, 301, 401, 501\}$; in this case, the concrete state graph for this query contains 16 states and 32 transitions. If *cid* ranged over an infinite data type, the concrete state graph would be infinite.

These definitions define the semantics of PARBAC policies but do not provide an effective algorithm for reachability analysis: parameters take values from an infinite type, so the concrete state graph is infinite, except for trivial policies.

4 Symbolic State Graph

This section defines symbolic states and symbolic transitions, which are the basis of our symbolic analysis algorithm.

A *symbolic state* is a pair (R, C) where R is a set of role instances (not necessarily concrete), and C is a constraint over variables that appear in R. A *constraint* is the constant *true* or a conjunction of tuple disequalities. A *tuple disequality* has the form $(e_1, \ldots, e_n) \neq (f_1, \ldots, f_n)$, where each e_i and f_i is a constant or a variable. We elide angle brackets around singleton tuples. Note that a conjunction of tuple disequalities is just a more compact notation for a logical combination of single (as opposed to tuple) inequalities, in conjunctive normal form.

For a constraint C, $\mathit{satisfiable}(C)$ is true if C does not contain a tuple disequality whose left and right sides are the same, and is false otherwise. For example, if C_0 denotes $X \neq cs$, then $\mathit{satisfiable}(C_0)$ is true, and $\mathit{satisfiable}(C_0[X \mapsto cs])$ is false. As another example, if C_1 denotes $(X, Y) \neq (Z, cs)$ then $\mathit{satisfiable}(C_1)$ and $\mathit{satisfiable}(C_1[X \mapsto Z])$ are true.

The above satisfiability test is correct when all variables range over infinite data types. If we extend the framework with a type system for parameters of role schemas, and the types may be finite, then the satisfiability test must be extended to check whether there are sufficiently many values of each type. This problem can easily be reduced to the graph coloring problem, which can be solved quickly for the small problem instances that typically arise in this setting.

A symbolic state (R, C) represents the concrete states obtained by instantiating R consistent with C; formally, the meaning of (R, C) is $[\![(R, C)]\!] = \{R\theta \mid \mathit{ground}(\theta) \wedge \mathit{satisfiable}(C\theta)\}$. For example, $(\{\mathit{Student}(\mathit{dept}{=}D)\}, D \neq cs)$ represents states containing a single instance of *Student* instantiated with any constant other than *cs*.

For a constraint C, $\mathit{simplify}(C)$ returns a new constraint obtained by removing tuple disequalities in which the two tuples have distinct constants in some component (such disequalities are equivalent to *true*, e.g., $(X, cs) \neq (Y, ee)$) and removing components of tuple disequalities that are equal in the two tuples (this yields a logically equivalent disequality, e.g., $(X, Y) \neq (X, Z)$ is replaced with $Y \neq Z$). If the last component is removed from a tuple disequality d (i.e., d becomes $() \neq ()$), then d simplifies to false. If all tuple disequalities in C are removed, C simplifies to *true*.

For a constraint C and a set *Vars* of variables, the projection of C on *Vars*, denoted $\mathit{project}(C, \mathit{Vars})$, is the constraint obtained from C by discarding disequalities that do not affect the satisfying values of variables in *Vars*. Specifically, $\mathit{project}(C, \mathit{Vars})$ constructs an undirected graph with a vertex for each tuple disequality in C, and with an edge between disequalities d_1 and d_2 if they share a variable (i.e., $\mathit{vars}(d_1) \cap \mathit{vars}(d_2) \neq \emptyset$, where $\mathit{vars}(e)$ is the set of variables that appear in expression e), and discards disequalities that are not reachable in the graph from any vertex d that mentions a variable in *Vars*. For example, $\mathit{project}(X \neq Y \wedge Y \neq Z \wedge U \neq V, \{Z\})$ equals $X \neq Y \wedge Y \neq Z$.

A substitution θ_1 is *more general than* a substitution θ_2, denoted $\theta_2 \preceq_g \theta_1$, if there exists a substitution θ such that $\theta_2 = \theta_1 \circ \theta$, where $\circ$ denotes composition. A *unifier* for role instances r_1 and r_2 is a substitution θ such that $r_1\theta = r_2\theta$. The *most general unifier* of r_1 and r_2, denoted $\mathit{mg\_unifier}(r_1, r_2)$, is a $\preceq_g$-maximal unifier for r_1 and r_2 (it is unique up to renaming of variables). For example, the substitution $[Y \mapsto cs, Z \mapsto X]$ is a most general unifier for $\rho(p = X, q = Y)$ and $\rho(p = Z, q = cs)$, while $[X \mapsto cs, Y \mapsto cs, Z \mapsto cs]$ is a less general unifier for them. For sets P_1 and P_2 of role instances, $\mathit{subset\_unifiers}(P_1, P_2) = \{\theta \in \mathit{Subst} \mid P_1\theta \subseteq P_2\theta\}$, and $\mathit{mg\_subset\_unifiers}(P_1, P_2)$ is the set of most general (i.e., $\preceq_g$-maximal) elements of $\mathit{subset\_unifiers}(P_1, P_2)$. For example, if $P_1 = \{\rho_1(cs)\}$ and $P_2 = \{\rho_1(X), \rho_1(Y), \rho_2(X)\}$, then $[X \mapsto cs]$ and $[Y \mapsto cs]$ are most general subset unifiers for P_1 and P_2, and $[X \mapsto cs, Y \mapsto cs]$ is a less general subset unifier for them.

The symbolic transition relation introduces *locally fresh* variables, i.e., variables not appearing in the source state of the transition that introduces them. Let $\mathit{freshSubst}(\theta, \mathit{vars}_1, \mathit{vars}_2)$ hold if θ maps variables in vars_1 to distinct

variables that are not in $vars_2$ and are chosen in some deterministic manner (e.g., choose the lexicographically smallest variables not in $vars_2$). To simplify the semantics of the graph, after the initial construction, we apply a straightforward, linear-time transformation *mkGloballyFresh* that renames introduced variables so they are globally fresh, i.e., each variable is introduced in at most one state in the graph.

Definition 5. The *symbolic transition relation* $T(URA, A)$ for a user-role administration policy URA and an assignment A of users to administrative roles contains a tuple $((R, C), (\varphi, \theta_f, \theta), (R', C'))$ if execution of rule φ in URA, instantiated with the substitution $\theta \circ \theta_f$, leads from symbolic state (R, C) to symbolic state (R', C'), where θ_f replaces variables in φ with fresh variables, and θ is a most general subset unifier of the positive preconditions of φ with roles in R. Formally, it is the least relation such that:

- $((R, C), (\varphi, \theta_f, \theta), (R', C')) \in T(URA, A)$ if $\varphi \in URA$ and $\varphi = can\_assign(r_a, (P, N), r_t)$ and there exist $R_p \subseteq R, (u_A, r_A) \in A$ such that
  - $freshSubst(\theta_f, vars(\varphi), vars((R, C)))$
  - $\theta \in mg\_subset\_unifiers(P\theta_f, R_p)$ (the roles in R_p satisfy the positive preconditions of φ) and $range(\theta) \subseteq vars(R_p) \cup Constants$.
  - $r_a\theta_f\theta = r_A$ (instantiating r_a yields the administrative role in A used to perform this role assignment)
  - $\theta(Self) = u_A$ (θ maps the distinguished variable *Self* to the identity u_A of the administrator performing this role assignment)
  - $R' = R\theta \cup \{r_t\theta_f\theta\}$
  - $distinct = \bigwedge_{r \in R \text{ such that } schema(r)=schema(r_t)} args(r\theta) \neq args(r_t\theta_f\theta)$ (the role being added is not already in the state; note that a conjunction with no conjuncts is *true*)
  - $neg = \bigwedge_{r_n \in N} \bigwedge_{r \in R \text{ such that } schema(r)=schema(r_n)} args(r\theta) \neq args(r_n\theta_f\theta)$ (the negative preconditions of φ are satisfied)
  - $C' = simplify(C\theta \wedge distinct \wedge neg)$
  - $satisfiable(C') = true$
- $((R, C), (\varphi, \theta_f, \theta), (R', C')) \in T(URA, A)$ if $\varphi \in URA$ and $\varphi = can\_revoke(r_a, r_t)$ and there exist $r \in R, \theta_f \in Subst, \theta \in Subst, (u_A, r_A) \in A$ such that
  - θ_f maps all variables in φ to distinct fresh variables, i.e., variables that do not appear in (R, C)
  - $\theta = mg\_unifier(r, r_t\theta_f)$ (r is the role instance being revoked) and $range(\theta) \cap range(\theta_f) = \emptyset$ (θ does not map variables in $vars(r) \cup vars(r_t\theta)$ to fresh variables in $range(\theta_f)$)
  - $r_a\theta_f\theta = r_A$ (instantiating r_a yields the administrative role in A used to perform this role assignment)
  - $\theta(Self) = u_A$ (θ maps the distinguished variable *Self* to the identity u_A of the administrator performing this role assignment)
  - $R' = R\theta \setminus \{r\theta\}$
  - $C_1 = simplify(C\theta)$
  - $satisfiable(C_1) = true$
  - $C' = project(C_1, vars(R'))$

Definition 6. The *symbolic state graph* for a user-role reachability query of the form in Definition 2 is a labeled directed graph $mkGloballyFresh(V, E)$, where the set V of vertices and the set E of edges are the smallest sets such that:

- $(\{r \mid (u_0, r) \in UA(\gamma) \wedge \neg admin(r)\}, true) \in V$.
- $((R, C), \varphi, (R', C')) \in E$ and $(R', C') \in V$ if $(R, C) \in V$ and there exist $\theta_f \in Subst$, $\theta \in Subst$, and $((R, C), (\varphi, \theta_f, \theta), (R', C')) \in T(URA, A)$.

Example 2. Consider the construction of the symbolic state graph for the query in Example 1. The initial state is $S_1 = (R_1, C_1) = (\{Student(dept = cs, cid = 501)\}, true)$. From S_1, the *can_assign* rule φ is applied (renaming D and CID to fresh variables D' and CID' respectively and then substituting D' with cs). This adds $TA(dept = cs, cid = CID')$ to the state under the constraint $CID' \neq 501$, resulting in a symbolic state $S_2 = (R_2, C_2) = (R_1 \cup \{TA(dept = cs, cid = CID')\}, (CID' \neq 501))$. S_2 represents the four concrete states $\{\{Student(dept = cs, cid = 501), TA(dept = cs, cid = X)\}$ for $X \in \{101, 201, 301, 401\}$. Similarly, from S_2, rule φ can be applied again (renaming D and CID to fresh variables D'_1 and CID'_1 respectively and then substituting D'_1 with cs). This leads to the state $S_3 = (R_3, C_3) = (R_2 \cup \{TA(dept = cs, cid = CID'_1)\}, C_2 \wedge (CID'_1 \neq 501) \wedge (CID'_1 \neq CID'))$. Repeating this process results in a symbolic state graph containing 5 states: S_1, S_2, S_3, $S_4 = (R_4, C_4) = (R_3 \cup \{TA(dept = cs, cid = CID'_2)\}, C_3 \wedge (CID'_2 \neq 501) \wedge (CID'_2 \neq CID') \wedge (CID'_2 \neq CID'_1))$ and $S_5 = (R_4 \cup \{TA(dept = cs, cid = CID'_3)\}, C_4 \wedge (CID'_3 \neq 501) \wedge (CID'_3 \neq CID') \wedge (CID'_3 \neq CID'_1) \wedge (CID'_3 \neq CID'_2))$, and 4 transitions: $S_1 \xrightarrow{\varphi} S_2 \xrightarrow{\varphi} S_3 \xrightarrow{\varphi} S_4 \xrightarrow{\varphi} S_5$. If *cid* ranges over an infinite data types, then the symbolic state graph would be infinite, because an infinite number of instances of $TA(dept = cs, cid = CID)$ would be added to the state.

The following theorems say that the symbolic transition relation and symbolic state graph are exact abstractions.

Theorem 1. Let A be a user-role assignment containing only administrative roles. Let URA be a user-role administration policy. For all symbolic states (R, C) and (R', C'), all policy rules φ in URA, and all substitutions θ_f and θ, $((R, C), (\varphi, \theta_f, \theta), (R', C')) \in T(URA, A)$ iff for all ground substitutions θ_c such that $satisfiable(C\theta_c) \wedge satisfiable(C'\theta_c)$, $(R\theta_c, (\varphi, \theta_c \circ \theta \circ \theta_f), R'\theta_c) \in T_c(URA, A)$.

Theorem 2. Let (V_c, E_c) and (V, E) be the concrete and symbolic state graphs, respectively, for a user-role reachability query of the form in Definition 2. (a) $\forall s_c \in V_c . \exists s \in V . s_c \in [\![s]\!]$ (all reachable concrete states are represented by reachable symbolic states). (b) $\forall s_c, s'_c, s, s', \varphi.\ s_c \in [\![s]\!] \wedge s'_c \in [\![s']\!] \wedge (s_c, \varphi, s'_c) \in E_c \Rightarrow (s, \varphi, s') \in E$ (all reachable concrete transitions are represented by reachable symbolic transitions). (c) $\forall s \in V.\ \exists s_c \in V_c.\ s_c \in [\![s]\!]$ (all reachable symbolic states represent reachable concrete states). (d) $\forall s, s', \varphi.\ (s, \varphi, s') \in E \Rightarrow \forall s_c \in [\![s]\!], s'_c \in [\![s']\!] . (s_c, \varphi, s'_c) \in E_c$ (all reachable symbolic transitions represent reachable concrete transitions).

5 Analysis Algorithm

This section presents a symbolic algorithm for user-role reachability analysis of PARBAC policies. The algorithm has two stages. The first stage performs a backward search from

the goal towards the initial state. However, some of the enabling conditions of the administrative actions are not checked during the backward search. In other words, this stage constructs an over-approximation of a backward slice (starting from the goal) of the symbolic state graph. The second stage determines which states in that graph are actually reachable, by running an exact forward search from the initial state, but limiting the search based on the results of stage 1. Compared to a purely forward algorithm, the backward stage improves the algorithm's efficiency by pruning the search space, and improves the algorithm's termination behavior. The overall strategy of using an approximate backward search followed by a forward search is reminiscent of Graphplan [5], although the details are quite different.

First Stage. The graph constructed by the first stage of the algorithm is an over-approximation for two reasons: negative preconditions are ignored, and disequality constraints are ignored. Negative preconditions could, at best, be only partially checked during the first stage, because the symbolic states constructed during the first stage might be *subsets* of the symbolic states that are actually reachable. This is because those states might actually contain additional roles that were needed to satisfy positive preconditions of earlier transitions (i.e., transitions between a state and the initial state); although some of those roles could perhaps be revoked, some of them might not be revocable by the administrative roles in A.

Since negative preconditions cannot be checked completely during the first stage, for simplicity, we do not check them at all during that stage; they are enforced during the second stage. Since disequality constraints are used primarily to enforce negative preconditions, we do not keep track of them during the first stage. Thus, each symbolic state in the *backward symbolic state graph* is simply a set of roles (i.e., role instances). Edges are determined by the *backward symbolic transition relation* T_b. A tuple $(R', (\varphi, \theta_f, \theta), R)$ is in that relation if a backward step from R' to R (i.e., R' is closer to the goal, and R is closer to the initial state) is possible—ignoring negative preconditions—using rule φ with the given substitutions, which are analogous to the substitutions in the forward symbolic transition relation introduced in Section 4. The backward symbolic transition relation considers only role assignment actions; it does not consider revocation, which cannot help satisfy positive preconditions.

Definition 7. The *backward symbolic transition relation* $T_b(URA, A)$ for a user-role administration policy URA and an assignment A of users to administrative roles is the least relation such that:

- $(R', (\varphi, \theta_f, \theta), R) \in T_b(URA, A)$ if $\varphi \in URA$ and $\varphi = can\_assign(r_a, (P, N), r_t)$ and there exist $r'_t \in R', P_1 \subseteq P, R_p \subseteq R' \setminus \{r'_t\}, (u_A, r_A) \in A$ such that
  - $freshSubst(\theta_f, vars(\varphi), vars(R'))$
  - $\theta \in mg\_subset\_unifiers(P_1\theta_f, R_p)$ (the positive preconditions in P_1 are satisfied by the roles in R_p; the other preconditions of φ will be added to R, acting as new sub-goals) and θ does not map variables in $vars(R')$ to locally fresh variables.
  - $r_a\theta_f\theta = r_A$ (instantiating r_a yields the administrative role in A used to perform this role assignment)
  - $\theta(Self) = u_A$ (θ maps the distinguished variable *Self* to the identity u_A of the administrator performing this role assignment)
  - $r'_t\theta = r_t\theta_f\theta$ (the role r'_t in R' is the role added by this transition)
  - $R = R'\theta \setminus \{r'_t\theta\} \cup \bigcup_{r \in P \setminus P_1} \{r\theta_f\theta\}$ (the earlier state R contains the roles in R', minus the role added by this transition, plus roles used to satisfy the remaining positive preconditions of φ)
  - $r'_t\theta \notin R$ (the role being added is not present in the earlier state R)

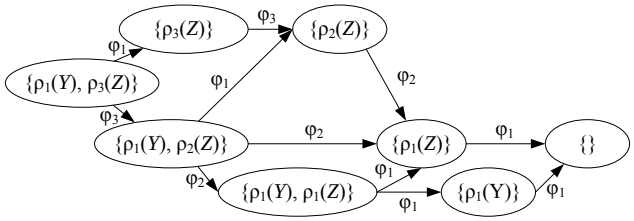

Figure 1: Backward symbolic graph for Example 3. An edge from R to R' labeled with φ means $(R', \varphi, R) \in E_b$. The roles all have one parameter, p, which we elide; e.g., $\rho_1(p = Y)$ is shown as $\rho_1(Y)$.

Definition 8. The *backward symbolic state graph* for a user-role reachability query of the form in Definition 2 is a labeled directed graph $mkGloballyFresh(V, E)$, where the sets V of vertices and E of edges are the smallest sets such that:

- $g \in V$.
- $(R', \varphi, R) \in E$ and $R \in V$ if $R' \in V$ and there exist $\theta_f \in Subst$ and $\theta \in Subst$ such that $(R', (\varphi, \theta_f, \theta), R) \in T_b(URA, A)$.

Example 3. The backward symbolic state graph for the following policy and query is shown in Figure 1 (for readability, the substitutions are omitted from the edge labels).
$RS = \{r_a, \rho_1(p), \rho_2(p), \rho_3(p)\}$, $UA(\gamma) = \{(u_a, r_a)\}$,
$A = \{(u_a, r_a)\}$, $g = \{\rho_1(p = Y), \rho_3(p = Z)\}$
$\varphi_1 = can\_assign(r_a,\ true,\ \rho_1(p = X))$
$\varphi_2 = can\_assign(r_a,\ \rho_1(p = X),\ \rho_2(p = X))$
$\varphi_3 = can\_assign(r_a,\ \rho_2(p = X) \wedge \neg\rho_1(p = X),\ \rho_3(p = X))$
$\varphi_4 = can\_revoke(r_a,\ \rho_1(p = X))$
$\varphi_5 = can\_revoke(r_a,\ \rho_2(p = X))$
$\varphi_6 = can\_revoke(r_a,\ \rho_3(p = X))$

Second Stage. The second stage performs a forward search and maintains a correspondence between states explored by the forward search, called *forward states*, and states explored during the first stage, called *backward states*. The correspondence is used to limit the forward search to explore only transitions that might be useful for reaching the goal. From each forward state (R, C) and each backward state R_b corresponding to it, the (unoptimized) forward algorithm explores (1) all enabled *can_assign* rules φ such that one of the backward states R_b corresponding to (R, C) is the target of an edge labeled with φ in the backward symbolic state graph, and (2) all enabled *can_revoke* rules. The resulting graph is called a *goal-directed forward symbolic state*

graph. Its nodes are pairs $((R, C), R_b)$ of a forward state (R, C) and a corresponding backward state R_b.

Definition 9. The *goal-directed forward symbolic state graph* for a user-role reachability query of the form in Definition 2 is a labeled directed graph $mkGloballyFresh(V, E)$, where (V_b, E_b) is the backward symbolic state graph for the query, and V and E are the smallest sets such that:

- $((UA_0, true), R_b) \in V$ for each $R_b \in V_b$ such that $(\exists \theta \in Subst.\ R_b\theta \subseteq UA_0)$, where UA_0, the initial role assignment for u_0, is given by $UA_0 = \{r \mid (u_0, r) \in UA(\gamma) \wedge \neg admin(r)\}$ (the initial forward state $(UA_0, true)$ is related to backward states that represent subsets of UA_0; intuitively, we use subset, instead of equality, because a backward state is a set of sub-goals, and we just require that the sub-goals are satisfied in the initial state)
- $(((R, C), R_b), (\varphi, \theta_f, \theta), ((R', C'), R'_b)) \in E$ and $((R', C'), R'_b) \in V$ if $((R, C), R_b) \in V$ and $((R, C), (\varphi, \theta_f, \theta), (R', C')) \in T(URA, A)$ and either φ is a *can_revoke* rule and $R'_b = R_b$, or φ is a *can_assign* rule and $(R'_b, \varphi, R_b) \in E_b$.

A forward state (R, C) satisfies goal g if there exists $\theta \in mg\_subset\_unifiers(g, R)$ such that $satisfiable(C\theta) = true$. The algorithm can easily provide a symbolic representation of all reachable instances of the goal: for each reachable forward state (R, C) that satisfies the goal, for each $\theta \in mg\_subset\_unifiers(g, R)$ such that $satisfiable(C\theta) = true$, add $(g\theta, C)$ to the result.

Termination. Termination is an issue, because the symbolic state graph may be infinite. For example, each use of the rule $can\_assign(Chair(dept = D), Faculty(dept = D), Instructor(dept = D, cid = C))$ introduces a fresh variable for the course identifier C, so a purely forward algorithm may add an unbounded number of distinct instances of the *Instructor* schema. The backward stage prevents divergence in many cases, but not all. Our algorithm is guaranteed to terminate if either (T1) the policy's positive-precondition dependency graph is acyclic, or (T2) all *can_assign* rules in the policy have at most one positive precondition. The *positive-precondition dependency graph* for a PARBAC policy is a directed graph that contains a vertex for each role schema and contains an edge from r_1 to r_2 if the policy contains a *can_assign* rule with r_1 in the positive precondition and r_2 in the target. The policies for both of our case studies satisfy both of these conditions, and we expect that most real policies satisfy at least one of them. The positive-precondition dependency graph is typically acyclic, because roles in the positive precondition in a *can_assign* rule are typically junior in the organizational hierarchy to the target role, and organizational hierarchies are acyclic. (T1) or (T2) directly ensure termination of the backward stage; this, in turn, ensures termination of the forward stage, provided the forward search is depth-first and is limited not to allow multiple transitions corresponding to the same backward edge on the search stack (i.e., forward paths that correspond to cycles in the backward graph are useless and can be pruned).

Optimizations. Our algorithm incorporates three optimizations. (1) A policy slicing transformation, similar to the one in [20] but enhanced with a forward pass, is applied before analysis. (2) In the second stage, a *can_assign* transition corresponding to a backward transition $(R'_b, \varphi, R_b) \in E_b$ is augmented so that it also revokes (*i.e.*, removes from the forward state) every revocable role r in the forward state that does not match any element of R_b (this is safe because those roles will not be needed to satisfy any preconditions in the rest of the path to the goal). A role r is *revocable* with respect to a user-role reachability query of the form in Definition 2 if A contains an administrative role with permission to revoke r. A role r_1 *matches* a role r_2 if r_1 and r_2 are instances of the same role schema and, for each parameter p of the schema, either (a) r_1 or r_2 has a variable as the value of p or (b) r_1 and r_2 contain the same constant as the value of p. This is a kind of partial-order reduction that performs revocations eagerly when it is safe to do so. (3) If the user wants only one reachable instance of the goal, then the forward search halts as soon as a state satisfying the goal is encountered; we call this *early stopping*.

Example 4. Consider the goal-directed forward symbolic graph for the policy and query in Example 3. The backward state $\emptyset$ corresponds to the initial forward state $(\{\}, true)$. Corresponding to the path $\{\rho_1(p = Z), \rho_3(p = Z)\} \overset{\varphi_1}{\to} \{\rho_3(p = Z)\} \overset{\varphi_3}{\to} \{\rho_2(p = Z)\} \overset{\varphi_2}{\to} \{\rho_1(p = Z)\} \overset{\varphi_1}{\to} \emptyset$ in the backward graph in Figure 1, the unoptimized algorithm constructs the following path in the goal-directed forward graph (for readability, the substitutions are omitted from the edge labels): $((\emptyset, true), \emptyset) \overset{\varphi_1}{\to} ((\{\rho_1(p = Z)\}, true), \{\rho_1(p = Z)\}) \overset{\varphi_2}{\to} ((\{\rho_1(p = Z), \rho_2(p = Z)\}, true), \{\rho_2(p = Z)\}) \overset{\varphi_4}{\to} ((\{\rho_2(p = Z)\}, true), \{\rho_2(p = Z)\}) \overset{\varphi_3}{\to} ((\{\rho_2(p = Z), \rho_3(p = Z)\}, true), \{\rho_3(p = Z)\}) \overset{\varphi_1}{\to} ((\{\rho_1(p = Y), \rho_2(p = Z), \rho_3(p = Z)\}, true), \{\rho_1(p = Y), \rho_3(p = Z)\})$. The last of these states satisfies the goal. With optimization (2), the *can_revoke* transition using rule φ_4 would be combined with the preceding *can_assign* transition using φ_2, and the transition that uses φ_3 to add $\rho_3(p = Z)$ would be extended to revoke $\rho_2(p = Z)$.

Fixed-Parameter Tractability. Expressing the complexity of the optimized backward algorithm as a function of the overall problem size alone is unsatisfactory, because the worst-case complexity with respect to this parameter is exponential, while we expect the typical complexity to be much better. To provide some insight into when and why this is the case, we express the complexity in terms of several metrics that characterize the "difficulty" of the policy. Our complexity results apply to policies that satisfy conditions (T1) and (T2) in the paragraph about termination.

Let G_b denote the backward symbolic state graph for a query. Each backward state R_b in G_b satisfies $|R_b| \leq |g|$, because each backward transition replaces the target role with the positive precondition of the selected *can_assign* rule. Let d_p denote the diameter of the positive-precondition dependency graph. Typically d_p is much smaller than $|RS|$, because it measures the height, not the total size, of an organization's administrative structure. The length of paths in G_b is bounded by $|g|d_p$, because each backward transition decreases the sum of the heights (in the positive precondition dependency graph) of the schemas of the roles in the backward state.

Let d_t denote the maximum number of *can_assign* rules with the same role schema as a target. The outdegree of a vertex in the backward state graph is bounded by $|g|d_t d_\theta$, where d_θ bounds the number of different successor states that can be reached from a given backward state using a given *can_assign* rule and different substitutions, i.e., it is

the maximum, over backward states R'_b in G_b and *can_assign* rules φ in the policy, of $|\{R_b \mid \exists \theta_f, \theta.\ (R'_b, (\varphi, \theta_f, \theta), R_b) \in T_b(URA, A)\}|$. Note that d_θ is bounded by $|R'_b|$ hence by $|g|$, because differences in θ that lead to differences in R_b come from matching the target of φ with different elements of R'_b. Thus, the outdegree of a vertex in the backward state graph is bounded by $|g|^2 d_t$. The number of nodes in a graph with maximum path length ℓ and maximum outdegree d is $O(d^\ell)$. Therefore, the number of backward states is $O((|g|^2 d_t)^{|g| d_p})$.

Let G_f denote the goal-directed forward symbolic state graph for the query. Every node in G_f is reachable by a simple path in G_f. Every simple path in G_f corresponds, by projection onto the second component of each node, to a distinct path in G_b, because (1) every transition in the goal-directed forward symbolic graph corresponds to execution of a backward symbolic transition that changes the second component (i.e., the backward state) in the node, and (2) distinct outgoing transitions from a state in the goal-directed forward symbolic graph must correspond to execution of different *can_assign* transitions hence to execution of different backward symbolic transitions. Furthermore, these paths in G_b contain at most one occurrence of each cycle in G_b, because transitions that go around a cycle in G_b a second time would not add more irrevocable roles or constraints to the corresponding forward states, hence the corresponding fragment of the path in G_f would be a cycle, contradicting the assumption that the path in G_f is simple. Therefore, the number of states in G_f is bounded by the number of paths in G_b that go around each cycle at most once. This is bounded by some function ϕ of the number of backward states. 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 generated state graph). Therefore, the worst-case time complexity of the overall backward algorithm is $O(|I|^c \phi((|g|^2 d_t)^{|g| d_p}))$, for some constant c and some function ϕ, where $|I|$ is the size of the problem instance (the query). This implies that user-role reachability for queries satisfying (T1) and (T2) is fixed-parameter tractable with respect to $\max(|g|, d_t, d_p)$. For the queries in our case studies, we found $|g| \leq 2$, $d_t \leq 10$, and $d_p \leq 3$.

6 Beyond Separate Administration

In [20], we presented two approaches to analysis of policies that do not satisfy separate administration. The first approach extends the algorithms to keep track of the user-role assignment for each administrator as well as the target user u_0; this is straightforward but may be computationally expensive. The second approach allows more efficient analysis of policies that satisfy an alternative assumption called *hierarchical role assignment*, which says, roughly, that each administrative role has authority to assign users only to selected roles that are junior to it in the role hierarchy. Both approaches can be adapted for analysis of PARBAC.

7 Case Studies

We used PARBAC policies for a university and a health-care facility as case studies. Unparameterized versions of these policies were used as case studies in [20]; those versions are unrealistic in the sense that they accommodate only one department, one course, one faculty, etc. The parameterized versions accurately handle multiple departments, multiple courses, multiple faculty, etc. Both policies have the following characteristics: (1) the positive-precondition dependency graph is acyclic; (2) every *can_assign* rule has at most one positive precondition; (3) for almost all *can_assign* rules, there is a corresponding *can_revoke* rule, so almost all roles are revocable; and (4) the policy does not satisfy separate administration, but hierarchical role assignment is satisfied for most sets of administrative roles. The policies contain about 3 dozen and 1 dozen *can_assign* rules, respectively.

University. Our PARBAC policy for a university controls assignment of users to student roles and employee roles. It contains 60 role schemas and 35 *can_assign* rules; expanding role hierarchy increases it to 625 rules. The role schemas include *Student*, *Undergrad*, *Undergrad*(*dept*), *Employee*, *Faculty*(*dept*), *Instructor*(*dept*, *cid*), etc. Role hierarchy relationships include *President* $\succeq$ *Provost* $\succeq$ *Dean* $\succeq$ *DeptChair* etc. A sample user-role reachability query is: Can an administrative user initially in *DeptChair*(*dept* = *cs*) add a user initially in role *Faculty*(*dept* = *ee*) to *QualExamCommittee*(*dept* = *cs*).

Health Care Facility. Our second case study is a PARBAC policy for a health care facility, based on policies in [10, 3]. The policy contains 14 *can_assign* rules and 2 *SMER* constraints. The role schemas include *Doctor*, *Doctor*(*patient*), *ReferredDoctor*(*patient*), *PrimaryDoctor*(*patient*), *Nurse*, *ThirdParty*, etc. Hierarchical role assignment is satisfied for most sets of administrative roles, but not as high a percentage of them as for the university policy.

8 Experimental Results

We implemented the symbolic algorithm described in this paper (including optimizations (1) and (3) but excluding optimization (2) and the search stack check described in the paragraph on termination; search stack checks are used in well-known efficient model checkers, such as SPIN, and should not significantly affect our performance results) and the forward and backward algorithms for analysis of unparameterized ARBAC in [20] using the XSB tabled logic programming system, version 3.1. We refer to the algorithms in [20] as *concrete algorithms*. Performance data in Table 1 were obtained on a 1.67 GHz Pentium Core 2 Duo machine with 2 GB RAM; other performance data were obtained on a 2GHz AMD Athlon machine with 1 GB RAM. The policies used in our experiments are available at `http://www.cs.stonybrook.edu/~stoller/parbac/`.

Case studies. We applied the symbolic algorithm to 5 user-role reachability queries for the university policy and 2 such queries for the health care policy (details are at the above URL). Each query is answered in less than 0.01 sec.

Performance Comparison for Parameterized Policies. These experiments evaluate the performance benefit of using symbolic analysis for parameterized policies in which all parameters range over finite types. These experiments compare the performance of the symbolic algorithm applied to a parameterized policy with the performance of the concrete algorithm applied to each unparameterized policy obtained by instantiating the parameterized policy using values from a single finite type, for varying sizes of the type.

To do this for a variety of "realistic" policies, we generate synthetic policies that are structurally similar to our university policy after expansion of role hierarchy and that contain about the same number of *can_assign* rules (namely, 625). The policies contain the same role schemas as the uni-

$\|g\|$	Symbolic			$\|T\|$	Concrete Backward		
			Node-1/Edge-1				
	Time	Mem	Node-2/Edge-2		Time	Mem	Node-1/Edge-1
1	0.00	2.86	27/291	1	0.00	2.86	27/291
			19/87	2	0.01	3.17	49/950
				3	0.01	3.88	76/2.2K
				4	0.02	5.13	109/4.4K
				5	0.05	7.16	149/7.7K
				6	0.10	10.08	194/12.5K
				7	0.19	14.16	246/19.1K
2	0.20	4.98	384/8.4K	1	0.05	3.77	370/7.9K
			808/10.0K	2	0.34	7.93	1.2K/47.5K
				3	1.40	20.61	3.0K/177.9K
				4	4.73	52.36	6.3K/511.6K
3	8.15	52.80	3.8K/125.1K	1	0.87	14.49	3.4K/108.6K
			27.9K/381K	2	10.66	117.13	20.4K/1.2M
				3	72.09	644.64	82.3K/7.3M

Table 1: Running time (sec) and memory consumption (MB) on parameterized policies. Node-i/Edge-i: number of nodes and edges generated in Stage i. K and M represent 10^3 and 10^6, respectively.

$\|g\|$	Symbolic		Concrete Backward		Concrete Forward	
	Time	Mem	Time	Mem	Time	Mem
1	0.00	0.35	0.00	0.36	0.00	0.43
2	0.00	0.37	0.00	0.38	0.02	0.61
3	0.02	0.54	0.01	0.56	0.03	0.67
4	0.29	2.78	0.25	2.96	0.03	0.71
5	1.81	10.29	1.26	10.03	0.04	0.77

Table 2: Running time (sec) and memory consumption (MB) on unparameterized policies.

versity policy. The number of *can_assign* rules per (target) role schema is chosen randomly following the distribution of rules per role schema in the university policy. The numbers of positive and negative preconditions per rule are chosen in an analogous way. For each rule, role schemas for the positive and negative preconditions are randomly selected and then instantiated based on the following observation about the university policy: in each rule, parameters (in different role schemas) are instantiated with same variable iff the parameters have the same name. "Easy" problem instances, for which policy slicing yields an empty policy, are discarded and replaced during policy generation.

Table 1 gives performance data for the algorithms. For the concrete backward algorithm, we report only numbers of nodes and edges in stage 1, because the nodes and edges in stage 2 are a subset of those in stage 1. Each data point is an average over 32 synthetic problem instances. We varied the size $|g|$ of the goal and the size $|T|$ of the finite type. Running time is rounded to the nearest 0.01 sec. We observe that the running time and memory consumption of the concrete backward algorithm grow quickly as a function of $|T|$. The symbolic algorithm outperforms the concrete backward algorithm when $|T| > 1$. Data for the concrete forward algorithm is omitted from Table 1 because that algorithm runs significantly slower than the other two algorithms when $|T| = 1$ and runs out of memory when $|T| > 1$, because (1) that algorithm is exponential in the number of mixed roles (roles that appear positively in some preconditions and negatively in others) and the number of mixed roles is large (namely, 18), and (2) that algorithm tends to generate states containing unnecessarily many distinct instances of some role schemas.

Performance Comparison for Unparameterized Policies. These experiments evaluate the performance penalty of unnecessarily using symbolic analysis on unparameterized policies, by comparing the performance of the symbolic and concrete algorithms applied to the same unparameterized policies. We used the synthetic unparameterized policies used for Tables 2(a) and 2(b) in [20] (running times in [20] were obtained with a different implementation of the concrete algorithms, in C++). Table 2 shows the time and space requirements of the three algorithms on the unparameterized policies used for Table 2(a) in [20], which contain 32 roles, including 5 mixed roles. Observe that the performance of the symbolic algorithm and concrete backward algorithm is similar for these policies. As expected, the time and memory consumption of these algorithms increases quickly with $|g|$, while the cost of the concrete forward algorithm increases slowly with $|g|$. All three algorithms terminate in less than 0.01 sec and consume less than 1 MB of memory on the unparameterized policies used for Table 2(b) in [20], where the number of roles varies from 100 to 500 and $|g| = 1$.

9 Related Work

Analysis of Unparameterized ARBAC Policies. A significant difference between this paper and prior papers on reachability analysis for ARBAC, including [14, 19, 20, 12], is that they consider only policies without parameters, so they are inapplicable to policies with parameters that range over infinite types, and they are inefficient when applied to policies with parameters over finite types that have been eliminated by exhaustive instantiation. Some general ideas in our prior work in [20] are also used here (e.g., backward search followed by forward search), but analysis of parameterized policies is significantly more difficult, requiring new algorithms and complexity results: the symbolic transition relations defined here are much more complicated than the concrete ones used in [20], the relationship between the two stages of the backward algorithm is different, new optimizations are needed for stage 2, different complexity parameters are used in the fixed-parameter tractability result, etc. Section 8 empirically compares our current work with [20].

Analysis of Other Kinds of Parameterized Systems. In general, parameterized systems have infinite state spaces, and the reachability problem for them is undecidable. Many specialized techniques have been developed for verification of various kinds of parameterized systems. There are two broad, overlapping classes of parameterized systems. In the first class, parameters represent the number of components (e.g., processes) in a system. There are numerous sound but incomplete reachability algorithms for such parameterized systems, e.g., [6, 8]. In the second class, parameters represent data values that range over infinite or unbounded domains. There are numerous techniques that analyze abstractions of such parameterized systems, giving up either soundness or completeness, e.g., [15]. In some restricted cases, such as data-independent systems and timed automata, sound and complete algorithms for reachability are known, e.g., [18, 1].

We are not aware of an existing symbolic reachability framework that can directly be applied for analysis of PARBAC, because of the following combination of features of the problem: (1) states are described by potentially unbounded sets of parameterized boolean variables corresponding to role membership facts (in other words, the number of roles in a

state, hence the number of state variables, is unbounded, as in the first category of parameterized systems described above), (2) the parameterized boolean variables may be used both positively and negatively in preconditions of transitions, (3) the parameters of the boolean variables range over an infinite data type (as in the second category of parameterized systems above), and (4) transitions may introduce an unbounded number of these parameters (*i.e.*, fresh variables) in the symbolic state graph (*cf.* the discussion of termination in Section 5). For example, work on verification of unbounded networks of processes, such as [9], is not applicable because it assumes that each process's state ranges over a fixed finite set. Work on verification of data-independent systems, such as [18], is not applicable because it assumes that fresh variables are not introduced during the search. Work on verification of cryptographic protocols, such as [4], allows unbounded numbers of nonces (represented by variables that range over an unbounded domain) but is not applicable because it does not allow these variables to be used in negative preconditions of transitions. Work on verification using inductive assertions, such as [2], heuristically constructs a candidate inductive invariant ϕ and calls a theorem prover to check whether ϕ is inductive and stronger than the property of interest, but is not applicable for a variety of reasons: the method can prove formulas containing universal but not existential quantifiers, so it can try to prove that a goal is unreachable but not that a goal is reachable (which would require existential quantification over the substitution, as in Definition 2), and if the heuristic method fails to prove that a goal is unreachable, we can draw no conclusions (in particular, we cannot conclude the goal is reachable); also, the heuristic cannot generate invariants that contain existential quantifiers, which are sometimes needed to accurately capture the effects of feature (4) above. Even if some existing parameterized verification framework could be applied, we would still need to define a symbolic transition relation for PARBAC, similar to Definition 5. Also, we are not aware of any fine-grained complexity results or fixed-parameter tractability results for such algorithms.

10 Conclusion

This paper presents an analysis algorithm for user-role reachability for ARBAC with parameters. Our algorithm is symbolic and does not need to consider all instantiations of the parameters. It exploits the structure of PARBAC policies, constructing a backwards graph to prune the search space. Future work on analysis for PARBAC includes algorithms guaranteed to terminate in more cases, efficient analysis of policies that do not satisfy separate administration or hierarchical role assignment, and analysis of policies that control changes to the role hierarchy. Future work beyond PARBAC includes analysis for trust management (e.g., [3]).

Acknowledgement. The authors thank Jason Crampton and the anonymous reviewers for valuable comments and suggestions, and Yogesh Upadhyay for implementing the conversion from hierarchical to non-hierarchical policies.

11 References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] T. Arons, A. Pnueli, S. Ruah, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Int'l. Conf. on Computer Aided Verification (CAV)*, 2001.

[3] M. Y. Becker. *Cassandra: Flexible Trust Management and its Application to Electronic Health Records*. PhD thesis, University of Cambridge, Oct. 2005.

[4] B. Blanchet and A. Podelski. Verification of cryptographic protocols: tagging enforces termination. *Theoretical Computer Science*, 333:67–90, Mar. 2005.

[5] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, Feb. 1997.

[6] E. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Trans. on Programming Languages and Systems*, 19(5):726–750, 1997.

[7] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

[8] E. Emerson and K. Namjoshi. Reasoning about rings. In *ACM Symposium on Principles of Programming Languages*, 1995.

[9] E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *International Conference on Automated Deduction*, 2000.

[10] M. Evered and S. Bögeholz. A case study in access control requirements for a health information system. In *Australasian Information Security Workshop*, 2004.

[11] L. Giuri and P. Iglio. Role templates for content-based access control. In *2nd ACM Workshop on RBAC (RBAC '97)*, pages 153–159, 1997.

[12] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough. Towards formal verification of role-based access control policies. *IEEE Trans. Dependable and Secure Computing*, 5(4):242–255, 2008.

[13] N. Li and Z. Mao. Administration in role based access control. In *Proc. ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS)*, pages 127–138, Mar. 2007.

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

[15] D. Long. *Model checking, abstraction and compositional verification*. PhD thesis, CMU, 1993.

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

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

[18] B. Sarna-Starosta and C. R. Ramakrishnan. Constraint-based model checking of data-independent systems. In *5th International Conference on Formal Engineering Methods (ICFEM)*, 2003.

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

[20] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *ACM Conference on Computer and Communication Security*, 2007.