Mining Parameterized Role-Based Policies

Zhongyuan Xu
Department of Computer Science
Stony Brook University, USA
zhoxu@cs.stonybrook.edu

Scott D. Stoller
Department of Computer Science
Stony Brook University, USA
stoller@cs.stonybrook.edu

ABSTRACT

Role-based access control (RBAC) offers significant advantages over lower-level access control policy representations, such as access control lists (ACLs). However, the effort required for a large organization to migrate from ACLs to RBAC can be a significant obstacle to adoption of RBAC. Role mining algorithms partially automate the construction of an RBAC policy from an ACL policy and possibly other information. These algorithms can significantly reduce the cost of migration to RBAC.

This paper defines a parameterized RBAC (PRBAC) framework in which users and permissions have attributes that are implicit parameters of roles and can be used in role definitions. Parameterization significantly enhances the scalability of RBAC, by allowing much more concise policies. This paper presents algorithms for mining such policies and reports the results of evaluating the algorithms on case studies. To the best of our knowledge, these are the first policy mining algorithms for a PRBAC framework. An evaluation on three small but non-trivial case studies demonstrates the effectiveness of our algorithms.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*; H.2.8 [**Database Management**]: Database Applications—*Data Mining*

Keywords: role mining; role-based access control

1. INTRODUCTION

Role-based access control (RBAC) offers significant advantages over lower-level access control policy representations, such as access control lists (ACLs). However, the effort required for a large organization to migrate from ACLs to RBAC can be a significant obstacle to adoption of RBAC. Role mining algorithms partially automate the construction of an RBAC policy from an ACL policy and possibly other information, such as user attributes. These algorithms can significantly reduce the cost of migration to RBAC.

Several versions of the role mining problem have been proposed. The most widely studied versions involve finding a minimum-size RBAC policy consistent with (i.e., equivalent to) given ACLs. Another important version of the problem arises when user attribute information is available. In this case, it is also desirable to maximize interpretability (also called "semantic meaning") of role membership with respect to the attribute information—in other words, to find roles whose membership can be characterized well using the attributes—and to minimize as policy size. Similarly, if permissions have attributes, interpretability of the set of permissions granted to each role can also be taken into account in an overall policy quality metric.

Allowing roles to have parameters significantly enhances the scalability of RBAC, by allowing much more concise policies. Parameterization is especially useful for expressing application-layer security policies. 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 policy statements parameterized by the class identifier or department name.

This paper defines an expressive parameterized RBAC (PRBAC) framework that supports a simple form of attribute-based access control (ABAC). In our framework, (1) users and permissions have attributes that are implicit parameters of roles, (2) the set of users assigned to a role is specified by an expression over user attributes, and (3) the set of permissions granted to a role is specified by an expression over permission attributes. We make role parameters implicit, rather than explicit, because it makes the framework and algorithms slightly simpler; our approach can easily be adapted to handle roles with explicit parameters. Every user and permission has an "id" attribute containing a unique name, so specifying the users and permissions associated with a role by enumeration, as in traditional RBAC, is a simple case of (2) and (3), respectively.

The main contribution of this paper is two algorithms for mining PRBAC policies from ACLs, user attributes, and permission attributes. To the best of our knowledge, it is the first policy mining algorithm for any parameterized RBAC framework or ABAC framework. At a high level, both algorithms work as follows. First, a conventional role mining algorithm is used to generate a set of candidate roles; attributes and parameterization are not considered in this step. For a policy like the above example, this step would

CODASPY'13, February 18–20, 2013, San Antonio, Texas, USA.

produce a separate role granting appropriate permissions to the chair of each department. Second, the algorithm attempts to form parameterized roles by merging sets of candidate roles from the first step; the resulting parameterized roles are added to the set of candidate roles. Containing the example, this step would form a parameterized role from the set of roles containing the chair role for each department. Third, the algorithm decides which of the candidate roles generated in the first two steps to include in the final policy. Inspired by [13], we consider two strategies for this. The *elimination* strategy repeatedly removes low-quality roles from the set of candidate roles, until no more roles can be removed without losing some of the permissions granted in the given ACL policy. The *selection* strategy repeatedly selects the highest-quality candidate role for inclusion in the PRBAC policy, until all permissions granted in the given ACL policy are granted by the PRBAC policy. For each of these two algorithms, we first present a simpler version that does not consider role hierarchy, and then present a version that generates hierarchical policies.

To evaluate whether these algorithms can successfully generate meaningful parameterized roles, we wrote three small but non-trivial PRBAC policies, generated ACL policies and attribute data from them, ran our algorithms on the resulting ACL policies and attribute data, and compared the mined PRBAC policies with the original policies. One of our algorithms successfully reconstructed the original PRBAC policies for all three case studies.

There are several directions for future work: applying the algorithms to more and larger case studies and developing better insight into when the algorithms succeed at producing intuitively desirable policies; adapting the algorithms to support a more conventional form of PRBAC in which role parameters are explicit; exploring algorithms for updating existing PRBAC policies, like StateMiner does for RBAC policies [10]; and exploring algorithms for mining ABAC policies. Our PRBAC framework already supports a simple form of ABAC. More thorough support for ABAC requires extending attribute expressions to allow membership tests for set-valued attributes, linear constraints for numerical attributes, etc., and then extending the policy mining algorithms to handle these features.

Section 2 defines our PRBAC framework. Section 3 defines the PRBAC mining problem. Section 4 presents our algorithms. Section 6 evaluates the algorithms on case studies. Section 7 discusses related work.

2. PARAMETERIZED RBAC (PRBAC)

PRBAC policies refer to attributes of users and permissions. User-attribute data is represented as a tuple $\langle U, A_U, f_U\rangle$, where U is a set of users, A_U is a set of user attributes, and f_U is a function such that $f_U(u,a)$ is the value of attribute a for user u. There is a special user attribute called uid that has a unique value for each user. This allows traditional identity-based roles to be represented in the same way as other roles. Similarly, permission-attribute data is represented as a tuple $\langle P, A_P, f_P\rangle$, where P is a set of permissions, A_P is a set of permission attributes, and f_P is a function such that $f_P(p,a)$ is the value of attribute a for permission p. Informally, a permission may be regarded as involving a resource and an operation, and a permission attribute may be an attribute of the resource or an attribute (i.e., argument) of the operation. There is a special permission attribute called pid that has a unique value for each permission. Let AttrVal be the set of all legal attribute values. We assume AttrVal includes a special value "$\perp$" that indicates that the value of an attribute is unknown.

Attribute expressions are used to express the sets of users and permissions associated with roles. A *conjunctive user-attribute expression* e_c is a function from user attributes A_U to $\text{Set}(\text{AttrVal} \setminus \{\perp\}) \cup \{\top\}$. The symbol $\top$ denotes the set of all legal values for an attribute. We say that expression e_c *uses* an attribute a if $e_c(a) \neq \top$. We refer to the set $e_c(a)$ as the conjunct for attribute a. A user u satisfies expression e_c, denoted $u \models e_c$, iff $(\forall a \in A_U : f_U(u,a) \in e_c(a))$. For example, if $A_U = \{\text{dept}, \text{level}\}$, the function e_c with $e_c(\text{dept}) = \{\text{CS}\}$ and $e_c(\text{level}) = \{\text{undergrad}, \text{grad}\}$ is a conjunctive user-attribute expression, which we write with syntactic sugar as $\text{dept} = \text{CS} \wedge \text{level} \in \{\text{undergrad}, \text{grad}\}$ (note that, when $e_c(a)$ is a singleton set $\{v\}$, we may write the conjunct as $a \in \{v\}$ or $a = v$). An *user-attribute expression* is a set, representing a disjunction, of conjunctive user-attribute expressions. A user u satisfies a user attribute expression e, denoted $u \models e$, iff $(\exists e_c \in e : u \models e_c)$. The meaning of a user-attribute expression e, denoted $[\![e]\!]_U$ is the set of users that satisfy it: $[\![e]\!]_U = \{u \in U \mid u \models e\}$. We say that a user-attribute expression e *characterizes* $[\![e]\!]_U$. We say that e *uses* an attribute a if some conjunctive user-attribute expression in e uses a. The definitions of *conjunctive permission-attribute expression* and *permission-attribute expression* are similar, except using the set A_P of permission attributes instead of the set A_U of user attributes. The meaning of a permission-attribute expression e, denoted $[\![e]\!]_P$ is the set of permissions that satisfy it: $[\![e]\!]_P = \{p \in P \mid p \models e\}$.

Constraints are used to express parameterization. Traditional PRBAC frameworks use explicit role parameters to indirectly express equalities between user attributes and permissions attributes; in our framework, such equalities are expressed directly, as constraints. For example, consider the policy that the chair of a department can update the course schedule for the department. This can be expressed using explicit role parameters by introducing a role chair(dept) and using a permission assignment rule such as PA(chair(dept), ⟨write, courseSchedule(dept)⟩). In our framework, we would define a chair role with the chairs of all departments as members, with permissions to write all course schedules, and with the constraint that the user's department equals the permission's department. The constraint ensures that each member of the role gets only the appropriate permissions. Informally, attributes used in the constraint act as role parameters.

A *constraint* is a set of equalities of the form $a_u = a_p$, where a_u is a user attribute and a_p is a permission attribute. User u and permission p satisfy constraint c, denoted $u, p \models c$, if for each equality $a_u = a_p$ in c, $f_U(u, a_u) = f_P(p, a_p)$.

A *core PRBAC policy* is a tuple $\langle U, P, R\rangle$ where U is a set of users, P is a set of permissions, and R is a set of roles, each represented as a tuple $\langle e_u, e_p, c\rangle$, , where e_u is a user-attribute expression, e_p is a permission-attribute expression, and c is a constraint. For a role $r = \langle e_u, e_p, c\rangle$, let $\text{uae}(r) = e_u$, $\text{pae}(r) = e_p$, and $\text{con}(r) = c$.

For example, the role $\langle \text{uid} = \{\text{Alice}, \text{Bob}\}, \text{operation} = \text{write} \wedge \text{resource} = \text{courseSchedule}, \text{dept} = \text{dept}\rangle$ has members Alice and Bob, has permissions to write course schedules for all departments (because the department attribute of the course schedule is not restricted by the permission-attribute expression), and has constraint dept = dept. If

$f_U(\text{Alice}, \text{dept}) = \text{CS}$ and $f_U(\text{Bob}, \text{dept}) = \text{EE}$, the constraint ensures that Alice only gets permission to write the CS course schedule, and Bob only gets permission to write the EE course schedule.

The user-permission assignment $\text{UPA}(\pi)$ induced by a policy π is defined by

$$\begin{aligned}
\text{asgndU}(r, U) &= \{u \in U \mid u \models \text{uae}(r)\} \\
\text{asgndP}(r, P) &= \{p \in P \mid p \models \text{pae}(r)\} \\
\text{asgndUP}(r, U, P) &= \{\langle u, p\rangle \in \text{asgndU}(r, U) \times \text{asgndP}(r, P) \mid \\
&\qquad u, p \models \text{con}(r)\} \\
\text{UPA}(\langle U, P, R\rangle) &= \textstyle\bigcup_{r \in R} \text{asgndUP}(r, U, P)
\end{aligned}$$

A *hierarchical PRBAC policy* is a tuple $\pi = \langle U, P, R, RH\rangle$, where U, P, and R are the same as in a core PRBAC policy, and the role hierarchy RH is an acyclic transitive binary relation on roles. A tuple $\langle r, r'\rangle$ in RH means that r is junior to r' (or, equivalently, r' is senior to r). This means that r inherits members from r', and r' inherits permissions from r. This is captured in the equations

$$\begin{aligned}
\text{ancestors}(r, R, RH) &= \{r' \in R \mid \langle r, r'\rangle \in RH\} \\
\text{descendants}(r, R, RH) &= \{r' \in R \mid \langle r', r\rangle \in RH\} \\
\text{authU}(r, U, R, RH) &= \text{asgndU}(r, U) \cup \\
&\qquad \bigcup_{r' \in \text{ancestors}(r, R, RH)} \text{asgndU}(r', U) \\
\text{authP}(r, P, R, RH) &= \text{asgndP}(r, P) \cup \\
&\qquad \bigcup_{r' \in \text{descendants}(r, R, RH)} \text{asgndP}(r', P)
\end{aligned}$$

The user-permission assignment $\text{UPA}(\pi)$ induced by a hierarchical PRBAC policy π is defined by:

$$\begin{aligned}
&\text{authUP}(r, U, P, R, RH) = \\
&\quad \{\langle u, p\rangle \in \text{authU}(r, U, R, RH) \times \text{authP}(r, P, R, RH) \mid \\
&\qquad u, p \models \text{con}(r)\} \\
&\text{UPA}(\langle U, P, R, RH\rangle) = \textstyle\bigcup_{r \in R} \text{authUP}(r, U, P, R, RH)
\end{aligned}$$

In the definition of $\text{authUP}(r, U, P)$, all authorized users and permissions of r, including the inherited ones, are subject to the constraint associated with r. However, constraints are not "inherited"; in particular, the constraint associated with a role r affects only r's contribution to the user-permission relation induced by the policy.

3. THE PROBLEM

A core PRBAC policy $\pi = \langle U, P, R\rangle$ is *consistent* with an ACL policy $\pi' = \langle U', P', UP'\rangle$ if $U = U'$, $P = P'$, and $\text{UPA}(\pi) = UP'$. A hierarchical PRBAC policy $\pi = \langle U, P, R, RH\rangle$ is *consistent* with an ACL policy $\pi' = \langle U', P', UP'\rangle$ if $U = U'$, $P = P'$, and $\text{UPA}(\pi) = UP'$.

A *policy quality metric* is a function from PRBAC policies to a totally-ordered set, such as the natural numbers. The ordering is chosen so that small values indicate high quality; this might seem counter-intuitive at first glance but is natural for metrics based on policy size.

The *core PRBAC policy mining problem* is: given an ACL policy π' and policy quality metric Q_{pol}, find a core PRBAC policy π that is consistent with π' and has the best quality, according to Q_{pol}, among policies consistent with π'. The *hierarchical PRBAC policy mining problem* is the same except that π is a hierarchical PRBAC policy.

Our algorithms aim to optimize the policy's *weighted structural complexity* (WSC), which is a generalization of policy size [8]. The weighted structural complexity of a core PRBAC policy is defined by

$$\begin{aligned}
\text{WSC}(e_c) &= \sum_{a \in \text{domain}(e_c)} e_c(a) = \top \;?\; 0 : |e_c(a)| \\
\text{WSC}(c) &= |c| \\
\text{WSC}(\langle e_u, e_p, c\rangle) &= w_1 \sum_{e_c \in e_u} \text{WSC}(e_c) + w_2 \sum_{e_c \in e_p} \text{WSC}(e_c) \\
&\qquad + w_3 \text{WSC}(c) \\
\text{WSC}(\langle U, P, R\rangle) &= \sum_{r \in R} \text{WSC}(r),
\end{aligned}$$

where $|s|$ is the cardinality of set s, and the w_i are user-specified weights. The weighted structural complexity WSC_{H} of a hierarchical PRBAC policy is defined in the same way, except with an additional term $w_4|RH|$, where the size of the role hierarchy RH is the number of tuples in it.

4. ALGORITHMS

This section presents our algorithms for the problems defined in Section 3.

4.1 Mining Core PRBAC Policies: Elimination Algorithm

Step 1: Generate Candidate Roles.

This step uses a traditional role mining algorithm to generate a set R_{can} of un-parameterized candidate roles without role hierarchy. Each role r in R_{can} is associated with a set $\text{asgndU}(r)$ of assigned users and a set $\text{asgndP}(r)$ of assigned permissions. We use CompleteMiner [11, 12] to generate candidate roles. Briefly, CompleteMiner generates a candidate role for every set of permissions that can be obtained by intersecting the sets of permissions granted to some users by the ACL policy. Note that CompleteMiner's goal is to include every reasonable candidate role in its output; CompleteMiner does not attempt to produce a minimum-sized policy.

We assume that no two candidate roles have exactly the same set of assigned users, and that no two candidate roles have exactly the same set of assigned permissions. This is true for the result of CompleteMiner and other standard role mining algorithms, because two roles with the same set of users or permissions can easily be merged into a single role.

Step 2: Generate Attribute Expressions for Candidate Roles.

This step computes minimum-sized attribute expressions that characterize the assigned users and assigned permissions of each candidate role, with preference given to (1) attribute expressions that do not use uid or pid, since attribute-based policies are generally preferable to identity-based policies, even when they have higher WSC, because attribute-based generalize better, and (2) conjunctive attribute expressions, because they are simpler than attribute expressions that use disjunction (in addition to conjunction).

Given a set s of users and the set U of all users, let $\text{minExpU}(s, U)$ be a minimum-sized user-attribute expression that characterizes s, subject to the preferences described

above. Given a set s of permissions and the set P of all permissions, let minExpP(s, P) be a minimum-sized permission-attribute expression that characterizes s, subject to the preferences described above. In both cases, at least one such attribute expression exists, because attributes uid and pid are present and have a unique value for each user or permission, respectively. For each $r \in R_{can}$, this step sets uae(r) = minExpU(asgndU$(r), U)$ and pae(r) = minExpP(asgndP(r), $P)$.

Our algorithm to compute minExpU(s, U) appears in Figure 1; the algorithm for minExpP is the same, except that A_U and f_U are replaced with A_P and f_P, respectively. The pseudocode for minExpU simply embodies the preferences described above. It uses an auxiliary function simplifyExp(e) that simplifies an attribute expression e by repeatedly looking for pairs of conjunctions c_1 and c_2 in e that differ in the value of a single attribute a and replacing c_1 and c_2 with a single conjunction c that agrees with c_1 and c_2 for all attributes except a and that maps a to $c_1(a) \cup c_2(a)$.

The pseudocode for minExpU also uses an auxiliary function minConjExpU that computes a minimum-sized conjunctive user-attribute expression that characterizes s, with preference given to attribute expressions that do not use uid. The first for-loop computes a conjunctive user-attribute expression e that attempts to characterize s without using uid. If this fails, then uid is needed to characterize s, and the algorithm returns a user-attribute expression that uses only uid. Otherwise, the algorithm uses e as a starting point for computation of a minimum-sized user-attribute expression for s that does not use uid. How could a smaller user attribute expression e' for s differ from e? It cannot be that some conjunct of e' is a strict subset of the corresponding conjunct of e, because then some user in s will not satisfy that conjunct. The only way that e' could differ from e is by replacement of some conjuncts with $\top$. The second for-loop considers all expressions that differ from e in this way.

Step 3: Generate Constraints for Candidate Roles.

We take con(r) to contain every equality that holds between every assigned user and every assigned permission of r. In other words, for each attribute a_u in A_U and each attribute a_p in A_P, we add the equality $a_u = a_p$ to con(r) iff $\forall u \in$ asgndU(r). $\forall p \in$ asgndP(r). $u, p \models a_u = a_p$. This is the strictest constraint that can be associated with r, because any stricter constraint would incorrectly eliminate some user-permission pairs in asgndUP(r, U, P). Using the strictest constraint for each role facilitates merging of roles in the next step.

Step 4: Merge Candidate Roles.

This step creates additional candidate roles by merging sets of candidate roles. A set s of roles is *mergeable* if there exists a role r' with the same assigned users, same assigned permissions, and same or larger user-permission assignment as the roles in s collectively, and asgndUP$(r', U, P) \subseteq UP'$, i.e., if there exists r' such that asgndU$(r') = \bigcup_{r \in s}$ asgndU(r) and asgndP$(r') = \bigcup_{r \in s}$ asgndP(r) and $\bigcup_{r \in s}$ asgndUP(r, U, P) $\subseteq$ asgndUP$(r', U, P) \subseteq UP'$. Assuming that all roles have distinct sets of assigned users and permissions (as mentioned in Step 1), s is mergeable only if there exists a constraint that can be associated with r' that prevents users assigned to one of the roles in s from incorrectly gaining permissions assigned to one of the other roles in s. Thus, to maximize the chance of a successful merge, we identify the strictest constraint that can be associated with r' and then check whether it prevents such gaining of permissions. The constraint associated with r' must not eliminate any user-permission assignment associated with any role in s. From Step 3, for each role r, con(r) is the strictest constraint that does not eliminate any user-permission assignment associated with r, so con(r') must be weaker (i.e., less strict) than or equal to con(r) for each r in s, so the strictest constraint that can be associated with r' is con$(r') = \bigcap_{r \in s}$ con(r). If the role r' with the assigned users, assigned permissions, and constraint specified above satisfies asgndUP$(r', U, P) \subseteq$

```
    function minExpU(s, U):
    // compute conjunctive and disjunctive user-attribute
    // expressions for s, and then compare them.
1:  e_c = {minConjExpU(s, U)}
2:  e_d = simplifyExp(⋃_{u∈s} minConjExpU(u, U))
3:  if e_c does not use uid and e_d uses uid
4:      return e_c
5:  end if
6:  if e_d does not use uid and e_c uses uid
7:      return e_d
8:  end if
9:  if WSC(e_c) ≤ WSC(e_d)
10:     return e_c
11: else
12:     return e_d
13: end if

    function minConjExpU(s, U):
    // check whether uid is needed to characterize s
    // in a conjunctive user-attribute expression.
14: for a in A_U \ {uid}
15:     e(a) = ⋃_{u∈s} f_U(u, a)
16:     if ⊥ ∈ e(a)
17:         e(a) = ⊤
18:     end if
19: end for
20: e(uid) = ⊤
21: if [[e]]_U ≠ s
        // uid is necessary (and sufficient) to characterize s
        // in a conjunctive user-attribute expression.
22:     return f_∅[uid ↦ ⋃_{u∈s} f_U(u, uid)]
23: end if
    // e characterizes s. check if there's a smaller conjunc-
    // tive user-attribute expression that characterizes s.
24: for each non-empty subset A of A_U \ {uid}
25:     e' = e[a ↦ ⊤ for a in A]
26:     if [[e']]_U = s
27:         if WSC(e') < WSC(e)
28:             e = e'
29:         end if
30:     end if
31: end for
32: return e
```

Figure 1: Algorithm to compute minExpU(s)**, where** s **is a set of users, and** U **is the set of all users.** $f[x \mapsto y]$ **denotes (a copy of) function** f **modified so that** $f(x) = y$**.** $f_\emptyset$ **denotes the empty function, i.e., the function whose domain is the empty set.**

```
1:  R' = R_can
    // R_mrg contains roles produced by merging.
2:  R_mrg = {}
    // for each r in R', remove r from R', then attempt to
    // merge r' with each remaining role in R' and each role
    // in R_mrg.
3:  for each r in R'
4:     R' = R' \ {r}
5:     for each r' in R' ∪ R_mrg
6:        r'' = merge({r, r'})
7:        if UPA(r'') ⊆ UP'
8:           R_mrg = R_mrg ∪ {r''}
9:        end if
10:    end for
11: end for
12: R_can = R_can ∪ R_mrg
```

Figure 2: Step 4 (Merge Candidate Roles) of elimination algorithm for core PRBAC policy mining.

UP' (note that $\bigcup_{r \in s} \text{asgndUP}(r, U, P) \subseteq \text{asgndUP}(r', U, P)$ holds by construction), then s is mergeable, and we set $\text{uae}(r') = \text{minExpU}(\text{asgndU}(r'), U)$ and $\text{pae}(r') = \text{minExpP}(\text{asgndP}(r'), P)$ and then add r' to R_{can} (note that we leave the roles in s in R_{can}); if not, s is not mergeable. Let $\text{merge}(s)$ denote the role r' defined above.

A simple algorithm for this step attempts to merge every subset s of R_{can}. We optimize the algorithm by exploiting a monotonicity property of merge, namely: $s \subseteq s'$ implies $\text{asgndUP}(\text{merge}(s), U, P) \subseteq \text{UPA}(\text{merge}(s'), U, P)$. Thus, if $\text{UPA}(\text{merge}(s)) \not\subseteq UP'$, the algorithm does not attempt to merge supersets s' of s, because $\text{UPA}(\text{merge}(s')) \not\subseteq UP'$ holds and hence s' is not mergeable. The algorithm also exploits the property $\text{merge}(s \cup \{r\}) = \text{merge}(\{\text{merge}(s), r\})$, which implies that arbitrary merges can be realized by merging in one role at a time. Pseudo-code for this step appears in Figure 2. The general structure of the code is similar to CompleteMiner [11, 12]. Our implementation incorporates another optimization, not shown in Figure 2. The order in which roles are merged does not affect the result, so we extend the algorithm to avoid merging the same roles in different orders. We define an arbitrary total order on roles. For each role r produced by merging, let $\text{maxMerge}(r)$ be the largest role used in the merges that produced r. In line 6, if r' was produced by merging, r is merged with r' only if $r > \text{maxMerge}(r')$.

Step 5 (Optional): Eliminate Unnecessary Constraints.

For a role r, an equality in $\text{con}(r)$ is *unnecessary* if removing it from $\text{con}(r)$ leaves $\text{asgndUP}(r)$ unchanged. This optional step removes each unnecessary equality from the constraint of each role in R_{can}.

Informally, one cannot tell from the given input whether to include unnecessary constraints in the PRBAC policy, because they do not affect consistency with the given ACL policy. Note that these "unnecessary" constraints may have been useful during merging, because they may help to prevent user-permission assignments from growing when roles are merged, but they provide no benefit after merging. The argument in favor of removing them (after merging) is to reduce policy size and hence increase policy quality. The argument in favor of keeping them is to be more conservative from the security perspective, specifically, to minimize the risk that the policy grants to a new user a permission that the new user should not have. This is related to how well the policy generalizes.

We consider this step as optional; in other words, the user decides whether unnecessary constraints should be removed.

Step 6: Eliminate Low-Quality Removable Candidate Roles.

This step eliminates low-quality removable candidate roles; the remaining roles form the generated PRBAC policy.

A *role quality metric* is a function $Q(r, U, P, R)$, where r is a new role whose quality is returned, U and P are the sets of users and permissions, respectively, R is the set of existing roles, and the range is a totally-ordered set. The ordering is chosen so that large values indicate high quality (note: this is opposite to the interpretation of the ordering for policy quality metrics).

Based on our goal of minimizing the generated policy's WSC, we define a role quality metric that assigns higher quality to roles with smaller WSC that cover more uncovered user-permission pairs; "uncovered" means that the user-permission pairs are not covered by roles in R. We capture this notion of quality using the ratio $\frac{|\text{uncovUP}(r,U,P,R)|}{\text{WSC}(r)}$ as the first component of the role quality metric, where $\text{uncovUP}(r, U, P, R)$ is the set of user-permission pairs covered by r and not covered by roles in R. Among roles with the same value of this ratio, we assign higher quality to roles that cover more user-permission pairs. We capture this by using $|\text{uncovUP}(r, U, P, R)|$ as the second component of the role quality metric, and ordering values of the role quality metric lexicographically. In summary, the role quality metric Q is defined as follows.

$$\text{uncovUP}(r, U, P, R) = \text{asgndUP}(r, U, P) \setminus \bigcup_{r' \in R} \text{asgndUP}(r', U, P)$$

$$Q(r, U, P, R) = \langle \tfrac{|\text{uncovUP}(r,U,P,R)|}{\text{WSC}(r)}, |\text{uncovUP}(r, U, P, R)| \rangle$$

A role r is *removable*, denoted $\text{removable}(r, U, P, R)$, if every user-permission pair covered by r is also covered by another role in R. Formally,

$$\text{removable}(r, U, P, R) = \text{asgndUP}(r, U, P) \subseteq \bigcup_{r' \in R \setminus \{r\}} \text{asgndUP}(r', U, P)$$

Pseudo-code for step 6 appears in Figure 3. In each iteration, R contains roles currently known to be in the result policy, and R_{can} contains roles that might later get added to the result policy. The algorithm evaluates removability of a role with respect to $R_{can} \cup R$ (instead of R), in order to minimize the set of roles considered unremovable; this leaves more roles in R_{can}, eligible for removal, and therefore leads to better policy quality. The algorithm evaluates role quality with respect to R, because this provides a better estimate of the role's quality in the final policy.

4.2 Mining Core PRBAC Policies: Selection Algorithm

Steps 1–5 of the selection algorithm for mining core PRBAC policies are the same as in the elimination algorithm for mining core PRBAC policies in Section 4.1. Step 6 is as follows.

1: $R = \emptyset$
2: while UPA($\langle U, P, R\rangle$) $\neq$ *UP*
// move unremovable roles from candidates R_{can}
// to result R.
3: $R_{unrm} = \{r \in R_{can} \mid \neg\text{removable}(r, U, P, R_{can} \cup R)\}$
4: $R_{can} = R_{can} \setminus R_{unrm}$
5: $R = R \cup R_{unrm}$
// discard the lowest-quality candidate role
6: if $\neg$empty(R_{can})
7: $r_{\min}$ = a role in R_{can} with minimal quality, *i.e.*,
8: $\forall r \in R_{can}.\ Q(r_{\min}, U, P, R) \leq Q(r, U, P, R)$.
9: $R_{can} = R_{can} \setminus \{r_{\min}\}$
10: end if
11: end while
12: return $\langle U, P, R\rangle$

Figure 3: Step 6 (Eliminate Low-Quality Removable Candidate Roles) of elimination algorithm for core PRBAC policy mining.

Step 6: Select Roles.

This step selects candidate roles for inclusion in the generated policy. It selects roles from highest quality to lowest, until every pair in the user-permission relation in the given ACL policy is covered. It uses the same role quality metric as Step 6 of the elimination algorithm in Section 4.1. Pseudo-code for this step is as follows.

1: $R = \emptyset$
2: while UPA($\langle U, P, R\rangle$) $\neq$ *UP*
3: $r_{\max}$ = a role in R_{can} with maximal quality, *i.e.*,
4: $\forall r \in R_{can}.\ Q(r_{\max}, U, P, R) \geq Q(r, U, P, R)$.
5: $R = R \cup \{r_{\max}\}$
6: $R_{can} = R_{can} \setminus \{r_{\max}\}$
7: end while
8: return $\langle U, P, R\rangle$

4.3 Mining Hierarchical PRBAC Policies: Elimination Algorithm

Steps 1–5 of this algorithm are the same as in the core PRBAC policy mining algorithm in Section 4.1. The remaining steps are as follows.

Step 6: Compute Role Hierarchy.

This step computes all possible role hierarchy relations between candidate roles. Let $r_1 \prec r_2$ if $r_1 \neq r_2 \wedge \text{asgndP}(r_1) \subseteq \text{asgndP}(r_2) \wedge \text{asgndU}(r_1) \supseteq \text{asgndU}(r_2)$. Let RH_{all} be the transitive reduction of $\prec$.

Step 7: Generate Result Policy.

This step starts by storing some current information about each candidate role in auxiliary data structures. Specifically, let $\text{authU}_0(r) = \text{asgndU}(r)$ and $\text{authP}_0(r) = \text{asgndP}(r)$ and $\text{authUP}_0(r) = \text{asgndUP}(r)$. Note that the assigned users and permissions might change as we generate the hierarchical policy, because some assigned users and permissions might become inherited instead. In contrast, the authorized users and permissions of each role r never change, always remaining equal to $\text{authU}_0(r)$ and $\text{authP}_0(r)$, respectively. Similarly, $\text{asgndUP}(r)$ might change, but $\text{authUP}(r)$ always remains equal to $\text{authUP}_0(r)$.

Our algorithm always generates policies with full inheritance [13]. This implies that a role hierarchy edge in RH_{all} is included in the result policy whenever the roles that it connects are included in the policy. Therefore, we associate with each candidate role the cost (WSC) of the edges that will be added to the policy if that role is added. We define a size metric on roles that reflects this: for a role r in R_{can}, and a set R of roles that have already been selected to be in the result policy,

$$\text{sizeof}(r, R, RH_{\text{all}}) = \text{WSC}(r) + \tfrac{w_4}{2}|\{r' \in R \mid \langle r, r'\rangle \in RH_{\text{all}} \vee \langle r', r\rangle \in RH_{\text{all}}\}|$$

The coefficient $\frac{w_4}{2}$ (recall that w_4 is introduced in Section 3) reflects that half of the cost of each inheritance relationship is attributed to each of the involved roles.

We define a role quality metric $Q_{\text{H}}(r, R, RH)$ similar to the metric in the non-hierarchical case, except using sizeof instead of WSC and using $\text{authUP}_0(r)$ instead of $\text{asgndUP}(r)$.

$$\text{uncovUP}_{\text{H}}(r, R) = \text{authUP}_0(r) \setminus \bigcup_{r' \in R} \text{authUP}_0(r')$$

$$Q_{\text{H}}(r, R, RH) = \frac{|\text{uncovUP}_{\text{H}}(r, R)|}{\text{sizeof}(r, RH)}$$

We define a function removable similar to the one in the non-hierarchical case, except using $\text{authUP}_0(r)$ instead of $\text{asgndUP}(r)$.

$$\text{removable}_{\text{H}}(r, R) = \text{authUP}_0(r) \subseteq \bigcup_{r' \in R \setminus \{r\}} \text{authUP}_0(r')$$

Pseudo-code for this step appears in Figure 5. It is similar to the pseudo-code in Figure 3 for Step 6 of the elimination algorithm for mining core PRBAC policies. The main difference is that this algorithm calls minExpU and minExpP to update $\text{uae}(r)$ and $\text{pae}(r)$, respectively, after roles have been added to the set R of roles that will be included in the generated policy. This reflects the fact that, in the presence of role hierarchy, we are free to choose $\text{asgndU}(r)$ in a way that minimizes the WSC of the policy, provided $\text{authU}(r, U, R, RH)$ remains equal to $\text{authU}_0(r)$, and similarly for $\text{asgndP}(r)$.

$\text{minExpU}_{\text{H}}(r, U, R, RH)$ returns a minimum-sized user attribute expression for r that excludes users that can be inherited from other roles in R along edges in RH if excluding those users reduces $\text{WSC}(\text{uae}(r))$; this is legitimate, because the sets of users assigned to and inherited by a role may overlap. Let $\text{inheritedU}(r, R, RH)$ denote the set of users that r inherits, i.e.,

$$\text{inheritedU}(r, R, RH) = \bigcup_{r' \in \text{ancestors}(r, R, RH)} \text{authU}_0(r').$$

Ideally, $\text{minExpU}_{\text{H}}(r, U, R, RH)$ would find a subset s of $\text{inheritedU}(r)$ that minimizes $\text{WSC}(\text{minExpU}(\text{authU}_0(r) \setminus s))$ and return $\text{minExpU}(\text{authU}_0(r) \setminus s)$. In practice, trying all subsets s of $\text{inheritedU}(r, R, RH)$ would be too slow. An obvious heuristic approximation is to try only the extrema—in other words, only $s = \emptyset$ and $s = \text{inheritedU}(r, R, RH)$. We adopt a heuristic approximation, shown in Figure 4 that is somewhat more thorough and correspondingly more expensive: it is exponential in the number of attributes, but polynomial in the number of users. Similar to our algorithm for minExpU in Figure 1, it starts by constructing an upper-bound expression e (for $\text{authU}_0(r)$) without using uid and then considers the expressions obtained setting to $\top$ the conjuncts of e corresponding to each subset of the attributes.

function $\text{minExpU}_\text{H}(r, U, R, RH)$:
// try to construct an expression representing a set
// in the required range, without using uid
for a in $A_U \setminus \{\text{uid}\}$
  $e(a) = \bigcup_{u \in \text{authU}_0(r)} f_U(u, a)$
  if $\bot \in e(a)$
    $e(a) = \top$
  end if
end for
$e(\text{uid}) = \top$
for each non-empty subset A of A_U
  $e' = e[a \mapsto \top \text{ for } a \text{ in } A]$
  // try to remove values from conjuncts of e'
  for a in $A_U \setminus A$
    for v in $e(a)$
      $e'' = e'[a \mapsto e'(a) \setminus \{v\}]$
      if $\text{isInRange}(e'', r, R, RH)$
        $e' = e''$
      end if
    end for
  end for
  if $\text{isInRange}(e', r, R, RH) \wedge \text{WSC}(e') < \text{WSC}(e)$
    $e = e'$
  end if
end for
if $\text{isInRange}(e, r, R, RH)$
  return e
end if
// uid is need to represent a set in the required range.
// choose the smallest set in that range, to get the
// smallest expression.
return $f_\emptyset[\text{uid} \mapsto \bigcup_{u \in \text{authU}_0(r) \setminus \text{inheritedU}(r,R,RH)} f_U(u, \text{uid})]$

// check whether $[\![e]\!]_U$ is in the required range
function isInRange(e,r,R,*RH*):
return $\text{authU}_0(r) \setminus \text{inheritedU}(r, R, RH) \subseteq [\![e]\!]_U$
        $\wedge\ [\![e]\!]_U \subseteq \text{authU}_0(r)$

Figure 4: Algorithm for $\text{minExpU}_\text{H}(r, U, R, RH)$.

However, instead of simply checking whether the resulting expression now denotes a larger set or still denotes the same set (namely, $\text{authU}_0(r)$) as in Fig 1, it exploits the flexibility that the expression may characterize any set between $\text{authU}_0(r) \setminus \text{inheritedU}(r)$ and $\text{authU}_0(r)$, by removing values from conjuncts of e (in order to make the denoted set smaller, partially counteracting the effect of setting some conjuncts to $\top$), provided the resulting expression still represents a superset of $\text{authU}_0(r) \setminus \text{inheritedU}(r)$, and then checks whether the resulting expression characterizes a set in the required range. If this fails to produce an expression representing a set in the required range, then an expression using uid is constructed.

$\text{minExpP}_\text{H}(r, P, R, RH)$ is defined similarly, except using $\text{inheritedP}(r, R, RH)$ instead of $\text{inheritedU}(r, R, RH)$, where $\text{inheritedP}(r, R, RH) = \bigcup_{r' \in \text{descendants}(r,R,RH)} \text{authP}_0(r')$.

Because we minimize $\text{WSC}(\text{uae}(r))$ and $\text{WSC}(\text{pae}(r))$ instead of $\text{asgndU}(r)$ and $\text{asgndP}(r)$, some inheritance relationships might become useless, if the users and permissions inherited by a role r through those relationships are also in $\text{asgndU}(r)$ and $\text{asgndP}(r)$, respectively. Such inheritance relationships could be eliminated without changing $\text{authUP}(r)$. We leave such relationships in the policy, because we want to generate policies with complete inheritance, as mentioned above. To illustrate the benefits of this approach, consider a problem instance in which there are user attributes indicating which users are employees (e.g., isEmployee = true) and which users are faculty (e.g., position = faculty), and that all faculty are employees. Suppose role mining produces roles corresponding to employee and faculty. If the assigned users of the employee role are characterized by isEmployee = true, then users in the faculty role are assigned users of the employee role, so an inheritance relationship between these roles is useless and could be eliminated, but this inheritance relationship is semantically meaningful and natural, so it is better to keep it in the policy.

$R = \emptyset$
while $\text{UPA}(\langle U, P, R, RH\rangle) \neq UP$
  // move unremovable roles from candidates R_{can}
  // to result R.
  $R_{unrm} = \{r \in R_{can} \mid \neg\text{removable}_\text{H}(r, R_{can} \cup R)\}$
  $R_{can} = R_{can} \setminus R_{unrm}$
  $R = R \cup R_{unrm}$
  // update uae and pae of candidate roles,
  // based on updated R
  for r in R_{can}
    $\text{uae}(r) = \text{minExpU}_\text{H}(r, R, RH_\text{all})$
    $\text{pae}(r) = \text{minExpP}_\text{H}(r, R, RH_\text{all})$
  end for
  // discard the lowest-quality candidate role
  if $\neg\text{empty}(R_{can})$
    r_min = a role in R_{can} with minimal quality, *i.e.*,
      $\forall r \in R_{can}.\ Q_\text{H}(r_\text{min}, R, RH) \leq Q_\text{H}(r, R, RH)$.
    $R_{can} = R_{can} \setminus \{r_\text{min}\}$
    remove tuples containing r_min from RH_all
  end if
end while
finalizePolicy(R, RH_all)

procedure finalizePolicy(R, RH_all):
// adjust uae and pae of roles in policy, based on final
// role hierarchy, to reduce WSC.
for r in R
  $\text{uae}(r) = \text{minExpU}_\text{H}(r, R, RH_\text{all})$
  $\text{pae}(r) = \text{minExpP}_\text{H}(r, R, RH_\text{all})$
end for
$RH = \{\langle r, r'\rangle \in RH_\text{all} \mid r \in R \wedge r' \in R\}$
return $\langle U, P, R, RH\rangle$

Figure 5: Step 6 (Eliminate Low-Quality Removable Candidate Roles) of elimination algorithm for hierarchical PRBAC policy mining.

4.4 Mining Hierarchical PRBAC Policies: Selection Algorithm

Steps 1–5 of this algorithm are the same as in the elimination algorithm for mining hierarchical PRBAC policies in Section 4.3. Step 6 is as follows.

Step 6: Select Roles.

1: $R = \emptyset$
2: while UPA($\langle U, P, R, RH \rangle$) $\neq UP$
// select the highest quality candidate role
3: $r_{\max}$ = a role in R_{can} with maximal quality, *i.e.*,
4: $\forall r \in R_{can}.\ Q_{\mathrm{H}}(r_{\max}, R, RH_{\mathrm{all}}) \geq Q_{\mathrm{H}}(r, R, RH_{\mathrm{all}})$.
5: $R = R \cup \{r_{\max}\}$
6: $R_{can} = R_{can} \setminus \{r_{\max}\}$
// update uae and pae of candidate roles,
// based on updated R
7: for r in R_{can}
8: uae(r) = minExpU$_{\mathrm{H}}(r, U, R, RH_{\mathrm{all}})$
9: pae(r) = minExpP$_{\mathrm{H}}(r, U, R, RH_{\mathrm{all}})$
10: end for
11: end while
12: finalizePolicy(R, RH_{all})

Figure 6: Step 6 (Select Roles) of selection algorithm for hierarchical PRBAC policy mining.

Pseudo-code for this step appears in Figure 6. It is similar to the pseudocode for Step 6 of the selection algorithm for mining core PRBAC policies in Section 4.2. The main differences are the addition of the for-loop to adjust the user-attribute expressions and permission-attribute expressions of previously selected roles when another role is selected, and the addition of the call to finalizePolicy at the end.

4.5 Complexity Analysis

This complexity analysis applies to all four of the above algorithms. The running time of CompleteMiner in Step 1 is worst-case exponential in $|P|$ but acceptable in practice, based on our experience applying it to small inputs in this work and larger inputs in previous work [13]. Let $R_{can}(i)$ denote the value of R_{can} after Step i; note that $|R_{can}(i)|$ is worst-case exponential in the size of the input policy. The running time of Step 2 is $O(|R_{can}(1)| \times (2^{|A_U|} + 2^{|A_P|}))$. The running time of Step 3 is $O(|R_{can}(2)| \times |A_U| \times |A_P|)$. The running time of Step 4 is $O(2^{|R_{can}(3)|})$, since every subset of $R_{can}(3)$ is explored in the worst case; however, the optimizations in Step 4 greatly reduce the number of explored subsets in our case studies. Steps 5, 6, and 7 are polynomial in $|R_{can}(4)|$, $|R_{can}(5)|$, and $|R_{can}(6)|$, respectively.

5. CASE STUDIES

This section describes the PRBAC policies we developed as case studies to illustrate our policy language and evaluate our algorithms.

The policies are written in a concrete syntax with the following kinds of statements. uae(r, e) associates conjunctive user-attribute expression e with role r. pae(r, e) associates conjunctive permission-attribute expression e with role r. The overall user attribute expression associated with role r is the disjunction of the expressions in the uae statements for r; similarly for the permission-attribute expression. con(r, c) associates constraint c with role r. rh(r, r') means that r is junior to r'. userAttrib$(u, a_1 = v_1, a_2 = v_2, \ldots)$ means that, for user u, attribute a_1 has value v_1, attribute a_2 has value v_2, etc.; uid $= u$ is implicit. permAttrib$(p, a_1 = v_1, a_2 = v_2, \ldots)$ is the analogous statement for permissions.

In each policy, we included only a few users in each "role instance", e.g., two or three users in each department. This provides sufficient data for the algorithm to discover the patterns, i.e., the parameterization. Increasing the number of users in each role instance only helps the algorithm, by providing stronger evidence for each pattern.

These case studies are small in size but non-trivial in structure. They includes roles with membership specified using uid, roles with membership specified using other attributes, roles with overlapping membership, roles with disjoint membership, roles with multiple pae statements, roles with constraints with multiple conjuncts, linear role hierarchy, diamond-shaped role hierarchy, etc.

University Case Study.

Our university case study controls access to gradebooks and course schedules. The policy appears in Figure 7, except that most of the userAttrib and permAttrib statements are omitted, to save space. For convenience, we give users names such as csStu1 and eeStu1, instead of Alice and Bob. User attributes include: position (student, faculty, or staff), dept (the user's academic or administrative department), crsTaken (course number of course being taken by a student; to keep the example small, we assume the student is taking at most one course, and it is in the student's department), crsTaught (course number of course taught by a faculty or TA; same assumptions as for crsTaken). Permission attributes include: resource (resource to which the operation is applied), operation (requested operation), dept (department to which the resource belongs), crsNum (number of the course that the resource is for), and student (student whose scores are read, for operation=readScoreStudent). The conjunct crsTaken=crsNum in the constraint for the Student role ensures that students can apply the readScoreStudent operation only to courses the student is taking. This is not essential, but it is natural and is advisable according to the defense-in-depth principle.

Healthcare Case Study.

Our healthcare case study controls access to items in electronic health records. The policy appears in Figure 8, except that most of the userAttrib and permAttrib statements are omitted, to save space. User attributes include: position (doctor or nurse; for other users, this attribute equals $\perp$); ward (the ward a patient or nurse is in), specialty (the medical specialty of a doctor), team (the medical team a doctor is in), and agentFor (the patient for which a user is an agent). Permissions for access to a health record have resource=HR ("HR" is short for "health record"). Other attributes of permissions for health records include: operation (the requested operation), patient (the patient that the HR is for), topic (the medical specialty to which the HR item is related), treatingTeam (the team of doctors treating the patient the HR is for), and ward (the ward housing the patient that the HR is for).

Engineering Department Case Study.

Our engineering department case study controls access to project-related documents. It is based on the running example in [9]. The policy appears in Figure 9, except that most of the userAttrib and permAttrib statements are omitted, to save space. The role hierarchy is a lattice: it has a diamond shape. User attributes include: dept (the user's department), project (the project the user is involved in; to keep the example small, we assume the user is involved in

```
// Student Role
uae(Student, position=student)
// Student can read his own scores in gradebook for course
// he is taking.
pae(Student, operation=readScoreStudent
            and resource=gradebook)
con(Student, dept=dept and crsTaken=crsNum
            and uid=student)

// Teaching Assistant (TA) Role
uae(TA, uid in {csStu2, eeStu2, csStu3, eeStu3})
// TA can add and read scores for any student in
// gradebook for course he/she is teaching.
pae(TA, operation in {addScore, readScore}
        and resource=gradebook)
con(TA, dept=dept and crsTaught=crsNum)

// Instructor Role
uae(Instructor, uid in {csFac1, csFac2, eeFac1, eeFac2})
// Instructor can change a score and assign a course grade
// in gradebook for course he/she is teaching.
pae(Instructor, operation in {changeScore, assignGrade}
                and resource=gradebook)
con(Instructor, dept=dept and crsTaught=crsNum)
rh(TA, Instructor)

// Department Chair Role
uae(Chair, uid in {csChair, eeChair})
// Chair can read and write course schedule for
// his/her department.
pae(Chair, operation in {read, write}
           and resource=courseSchedule)
// Chair can assign grades for courses in his/her
// department.
pae(Chair, operation=assignGrade and resource=gradebook)
con(Chair, dept=dept)

// Registrar Role
uae(Registrar, dept=registrar)
// Staff in registrar's office can modify course schedules
// for all departments.
pae(Registrar, operation=write and resource=courseSchedule)

// User Attribute Data. The userAttrib statement for one
// user is shown here; the full policy contains 19 users.
userAttrib(csStu1, position=student, dept=cs, crsTaken=101)

// Permission Attribute Data. The permAttrib statement
// for one permission is shown here; the full policy
// contains 26 permissions.
permAttrib(cs101addScore, dept=cs, crsNum=101,
   operation=addScore, resource=gradebook)
```

Figure 7: University case study

```
// Nurse Role
uae(Nurse, position=nurse)
// Nurse can read and add HR items with topic=general
// for patients in his/her ward.
pae(Nurse, resource=HR and operation in {readItem, addItem}
           and topic=general)
con(Nurse, ward=ward)

// Doctor Role
uae(Doctor, position=doctor)
// Doctor can read and add HR items related to his specialty
// for patients being treated by his/her team.
pae(Doctor, resource=HR
            and operation in {readItem, addItem})
con(Doctor, team=treatingTeam and specialty=topic)

// Patient Role
uae(Patient, uid in {oncPat1, oncPat2, carPat1, carPat2})
// A patient can read and add items with topic=patientNote
// in his/her HR.
pae(Patient, resource=HR
             and operation in {readItem, addItem}
             and topic=patientNote)
con(Patient, uid=patient)

// Agent Role
uae(Agent, uid in {agent1, agent2})
// Agent can add an item with topic=agentNote in HR
// for patient whose agent he/she is.
pae(Agent, resource=HR and operation=addItem
     and topic=agentNote)
// Agent can read an item with topic patientNote or
// agentNote in HR for patient whose agent he/she is.
pae(Agent, resource=HR and operation=readItem
     and topic in {patientNote, agentNote})
con(Agent, agentFor=patient)

// User Attribute Data. The userAttrib statement for one
// user is shown here; the full policy contains 14 users.
userAttrib(oncNurse1, position=nurse, ward=oncWard)

// Permission Attribute Data. The permAttrib statement
// for one permission is shown here; the full policy
// contains 24 permissions.
permAttrib(rdOncItemOncPat1, resource=HR,
  operation=readItem, patient=oncPat1, topic=oncology,
  treatingTeam=oncTeam1, ward=oncWard)
```

Figure 8: Healthcare case study

at most one project, and it is in the user's department), and specialty (the user's specialty, e.g., testing). Permission attributes include: resource (resource to which the operation is applied), operation (requested operation), dept (department to which the resource belongs), and project (project to which the resource belongs).

6. EVALUATION

This section describes an evaluation of the effectiveness of our algorithms, based on the case studies in Section 5. For each case study, we generated an equivalent ACL policy and an attribute data file from the PRBAC policy, ran our hierarchical PRBAC policy mining algorithms on the resulting ACL policy and attribute data, and then compared the generated PRBAC policy to the original PRBAC policy.

The same methodology could be applied starting with synthetic (i.e., pseudo-randomly generated) PRBAC policies. We did not do this, for two reasons. First, it is difficult to

```
// Engineer Role
// In this example, all users are engineers.
uae(Engineer, true)
// Engineer can read the project plan and test plan
// for the project he/she is working on.
pae(Engineer, operation=read
              and resource in {projectPlan, testPlan})
con(Engineer, dept=dept and project=project)

// ProductionEngineer Role
uae(ProductionEngineer, specialty=production)
// Production Engineer can write the project plan
// for the project he/she is working on.
pae(ProductionEngineer, operation=write
                        and resource=projectPlan)
con(ProductionEngineer, dept=dept and project=project)
rh(Engineer, ProductionEngineer)

// QualityEngineer Role
uae(QualityEngineer, specialty=testing)
// Quality Engineer can write the test plan for the
// project he/she is working on.
pae(QualityEngineer, operation=write and
                     resource=testPlan)
con(QualityEngineer, dept=dept and project=project)
rh(Engineer, QualityEngineer)

// ProjectLead Role
uae(ProjectLead, specialty=management)
// Project Lead can create a budget for the project
// he/she is leading.
pae(ProjectLead, operation=create and resource=budget)
con(ProjectLead, dept=dept and project=project)
rh(ProductionEngineer, ProjectLead)
rh(QualityEngineer, ProjectLead)

// User Attribute Data. The userAttrib statement for one
// user is shown here; the full policy contains 14 users.
userAttrib(qe1, dept=ads, project=alpha, specialty=testing)

// Permission Attribute Data. The permAttrib statement
// for one permission is shown here; the full policy
// contains 10 permissions.
permAttrib(rpa1, dept=ads, project=alpha, operation=read,
    resource=projectPlan)
```

Figure 9: Engineering department case study

generate "realistic" synthetic policies, so effectiveness of our algorithm on synthetic policies might not be representative of its effectiveness on real policies. Second, it is difficult to evaluate the effectiveness of our algorithms on synthetic policies: in case of differences between the synthetic policy and the mined policy, there would be no basis for determining which one is better (for example, the synthetic policy might be unnecessarily complicated, and the mined policy might be better). We could determine which policy has lower WSC, but minimizing WSC is just a heuristic aimed at helping the algorithm discover high-level structure, and we do not know what the best high-level structure is for synthetic policies. Ideally, we would evaluate the algorithms on access control policies in actual use, but we do not know of any publicly available deployed access control policies with accompanying attribute data.

Case Study	$\|U\|$	$\|P\|$	$\|UP\|$	$\|A_U\|$	$\|A_P\|$	$\|R_{can}\|$	$\|R\|$	Time
university	19	26	42	4	5	203	5	2.1 1.2
healthcare	14	24	42	5	6	42	4	.55 .43
eng. dept.	14	10	42	3	4	24	4	.21 .19

Figure 10: Running times and size metrics for case studies.

In summary, for all three case studies, the selection algorithm for mining hierarchical PRBAC policies, without optional Step 5 (Eliminate Unnecessary Constraints), successfully reconstructs the original PRBAC policy from the ACLs and attribute data.

We implemented the algorithms in Java and ran them on a laptop with an Intel Core i3 2.13 GHz CPU. In our experiments, all weights w_i in the definition of WSC are equal to 1. Table 10 shows, for each case study, several size metrics and the running times of both algorithms. The "$|R_{can}|$" column contains the size of R_{can} after Step 4. The "Time" column contains the running times (in seconds) for the elimination and selection algorithms, respectively, for mining hierarchical PRBAC policies and including the optional Step 5. We also measured the running time of each step. In all cases except one, Step 4 is the most expensive step; the one exception is the elimination algorithm on the university case study, for which Step 7 is the most expensive step.

Results of university case study.

We ran the elimination and selection algorithms on ACLs and attribute data generated from the university case study. Without Step 5 (Eliminate Unnecessary Constraints), the selection algorithm reconstructs the original PRBAC policy. The elimination algorithm does slightly worse, producing two roles, corresponding to TAs for CS101 and CS601, instead of a single parameterized TA role. With Step 5 (Eliminate Unnecessary Constraints), the output of the elimination algorithm stays the same, and the output of the selection algorithm becomes the same as the output of the elimination algorithm.

Results of healthcare case study.

We ran the elimination and selection algorithms on ACLs and attribute data generated from the healthcare case study. Without Step 5 (Eliminate Unnecessary Constraints), both algorithms reconstruct the original PRBAC policy. With Step 5 (Eliminate Unnecessary Constraints), the elimination algorithm still reconstructs the original PRBAC policy, but the selection algorithm does slightly worse, producing two roles, corresponding to cardiologists and oncologists, instead of a single parameterized Doctor role.

Results of engineering department case study.

We ran the elimination and selection algorithms on ACLs and attribute data generated from the engineering department case study. The selection algorithm reconstructs the original PRBAC policy. The elimination algorithm reconstructs the ProductionEngineer and ProjectLead roles, but each of the other two roles in the resulting policy contain some general engineers and some quality engineers. For both

algorithms, the results are unaffected by Step 5 (Eliminate Unnecessary Constraints).

Limitations.

Our algorithm does not reconstruct the original policy for some variants of the health care case study, because CompleteMiner does not generate the candidate roles that need to be merged to produce the original roles. For example, suppose we modify the policy so that a patient's agent has all permissions of that patient, plus some agent-specific permissions. As a result, the agent's permissions are a superset of the patient's permissions, and the roles generated by CompleteMiner all have the property that, if the role contains the patient, then it also contains the agent. This prevents subsequent steps of the algorithm from discovering a parameterized patient role, because different constraints are needed for patients and agents, as one can see from the patient and Agent roles in Figure 8. To overcome this limitation, Step 1 should be extended to take attribute information into account when generating candidate roles.

7. RELATED WORK

As mentioned in Section 1, we are not aware of any prior work on policy mining for PRBAC or ABAC. Our policy mining algorithms build on two pieces of prior work on role mining for RBAC: Vaidya, Atluri, and Warner's CompleteMiner algorithm for generating candidate roles [11, 12], and Xu and Stoller's elimination and selection algorithms for deciding which candidate roles to include in the final policy [13]. The novel part of our algorithms are the middle steps, in which constraint generation and role merging are used to discover parameterization.

Xu and Stoller's elimination algorithm is partly inspired by Molloy *et al.*'s Hierarchical Miner algorithm for mining roles with semantic meaning based on user-attribute data [8]. Colantonio *et al.* developed a different method for taking user-attribute data into account during role mining; their method partitions the set of users based on the values of selected attributes, and then performs role mining separately for each of the resulting sets of users [2].

We use role quality and policy quality metrics based on weighted structural complexity [8]. Other role quality and policy quality metrics have been proposed. Colantonio *et al.* proposed metrics that measure how well roles fit the hierarchical structures of an organization and its business processes [1]. These metrics could be incorporated in our algorithm. Qi *al.* proposed a metric for optimality of role hierarchies and an efficient heuristic algorithm for mining role hierarchies based on that metric [5]. Their work could be extended to accommodate parameters and combined with our approach to discovering parameterized roles.

Several access control frameworks that support some form of parameterized roles have been proposed. The earliest ones are by Giuri and Iglio [4] and Lupu and Sloman [7]; the role templates and policy templates, respectively, in these frameworks support parameterized roles. More recently, Ge and Osborn [3] and Li and Mao [6] proposed RBAC frameworks with parameterized roles. The most visible difference between parameterization in these frameworks and ours is that role parameters are explicit in these frameworks but implicit in ours. However, this difference is more superficial than significant: our approach to PRBAC policy mining can be adapted to PRBAC frameworks with explicit parameters.

8. ACKNOWLEDGEMENTS

This material is based upon work supported by ONR under Grant N00014-07-1-0928, NSF under Grant CNS-0831298, and AFOSR under Grant FA0550-09-1-0481.

9. REFERENCES

[1] A. Colantonio, R. Di Pietro, A. Ocello, and N. V. Verde. A formal framework to elicit roles with business meaning in RBAC systems. In *Proc. 14th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 85–94, 2009.

[2] A. Colantonio, R. Di Pietro, and N. V. Verde. A business-driven decomposition methodology for role mining. *Computers & Security*, 2012.

[3] M. Ge and S. L. Osborn. A design for parameterized roles. In *Research Directions in Data and Applications Security XVIII, IFIP TC11/WG 11.3 Eighteenth Annual Conference on Data and Applications Security*, pages 251–264. Kluwer, 2004.

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

[5] Q. Guo, J. Vaidya, and V. Atluri. The role hierarchy mining problem: Discovery of optimal role hierarchies. In *Proc. 2008 Annual Computer Security Applications Conference (ACSAC)*, pages 237–246. IEEE Computer Society, 2008.

[6] 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. ACM Press, Mar. 2007.

[7] E. Lupu and M. Sloman. Reconciling role based management and role based access control. In *Proc. 2nd ACM Workshop on Role Based Access Control (RBAC'97)*, pages 135–141, November 1997.

[8] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. B. Calo, and J. Lobo. Mining roles with multiple objectives. *ACM Trans. Inf. Syst. Secur.*, 13(4), 2010.

[9] R. S. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135, 1999.

[10] H. Takabi and J. B. D. Joshi. StateMiner: an efficient similarity-based approach for optimal mining of role hierarchy. In *Proc. 15th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 55–64, 2010.

[11] J. Vaidya, V. Atluri, and J. Warner. RoleMiner: Mining roles using subset enumeration. In *Proc. 13th ACM Conference on Computer and Communications Security (CCS)*, pages 144–153, 2006.

[12] J. Vaidya, V. Atluri, J. Warner, and Q. Guo. Role engineering via prioritized subset enumeration. *IEEE Trans. Dependable Secur. Comput.*, 7(3):300–314, 2010.

[13] Z. Xu and S. D. Stoller. Algorithms for mining meaningful roles. In *Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 57–66, 2012.