Algorithms for Mining Meaningful Roles*

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, such as user attributes. These algorithms can significantly reduce the cost of migration to RBAC.

This paper proposes new algorithms for role mining. The algorithms can easily be used to optimize a variety of policy quality metrics, including metrics based on policy size, metrics based on interpretability of the roles with respect to user attribute data, and compound metrics that consider size and interpretability. The algorithms all begin with a phase that constructs a set of candidate roles. We consider two strategies for the second phase: start with an empty policy and repeatedly add candidate roles, or start with the entire set of candidate roles and repeatedly remove roles. In experiments with publicly available access control policies, we find that the elimination approach produces better results, and that, for a policy quality metric that reflects size and interpretability, our elimination algorithm achieves significantly better results than previous work.

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. However, interpretability of roles is also crucial, because typically, a role produced by a role mining algorithm will be adopted by security administrators only if they can identify a reasonable interpretation of the role, in which case the role is said to be "meaningful". Indeed, researchers at HP Labs wrote that "the biggest barrier we have encountered to getting the results of role mining to be used in practice" is that "customers are unwilling to deploy roles that they can't understand." [3]. When data about attributes of users is available, it can be used to help identify meaningful roles. The general idea is that a role is meaningful if its set of members can be characterized by an expression involving user attributes. There are numerous reasonable variants of the definitions of policy size and interpretability, and different definitions may be appropriate in different contexts.

The main contribution of this paper is a role mining algorithm that can easily be used to optimize a variety of policy quality metrics—including metrics based on policy size, metrics based on interpretability of the roles with respect to user attribute data, and compound metrics that consider size and interpretability—and that achieves good results.

All of our algorithms begin with a phase that constructs a set of candidate roles. We consider two strategies for the second phase: start with an empty policy and repeatedly add candidate roles, or start with the entire set of candidate roles and repeatedly remove roles. In experiments with publicly available access control policies, we find that the elimination approach produces better results, and that, for a previously proposed policy quality metric that reflects size and interpretability, our elimination algorithm achieves significantly better results than previous work that aims to optimize that metric, even though our algorithm is not specifically tuned for that metric.

Other contributions of this paper include:

- an investigation of the effect of varying the order in which roles are considered for removal in the elimination algorithm;

*This work was supported in part by ONR under Grant N00014-07-1-0928, NSF under Grant CNS-0831298, and AFOSR under Grant FA0550-09-1-0481.

SACMAT'12, June 20–22, 2012, Newark, New Jersey, USA.

- an algorithm for synthesizing user attribute data. The use of synthetic user attribute data in experiments is regrettable but currently unavoidable, due to the lack of publicly available real user attribute data.

2. PROBLEM DEFINITION

This section defines the role mining problems that we consider. Our definitions are similar to those in [9].

Policies and Policy Quality.

An *ACL policy* is a tuple $\langle U, P, UP\rangle$, where U is a set of users, P is a set of permissions, and $UP \subseteq U \times P$ is the user-permission assignment.

An *RBAC policy* is a tuple $\langle U, P, R, UA, PA, RH\rangle$, where R is a set of roles, $UA \subseteq U \times R$ is the user-role assignment, $PA \subseteq R \times P$ is the permission-role assignment, and $RH \subseteq R \times R$ is the role inheritance relation. Specifically, $\langle r, r'\rangle \in RH$ means that r is senior to r', hence all permissions of r' are also permissions of r, and all members of r are also members of r'.

An *RBAC policy with direct assignment* is a tuple $\langle U, P, R, UA, PA, RH, DA\rangle$, which is an RBAC policy extended with a direct user-permission assignment $DA \subseteq U \times P$. Allowing direct assignment of permissions to users provides more flexibility to handle anomalous permissions.

An RBAC policy is *consistent* with an ACL policy if $UA \circ PA = UP$, where $\circ$ is composition of relations. An RBAC policy with direct assignment is *consistent* with an ACL policy if $UA \circ PA \cup DA = UP$.

User-attribute data is a tuple $\langle A, f\rangle$, where A is a set of attributes, and f is a function such that $f(u, a)$ is the value of attribute a for user u. For simplicity, we assume that all attribute values are natural numbers.

A *policy quality metric* is a function from RBAC policies (or RBAC policies with direct assignment) 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 it is natural for metrics such as policy size. We define two basic policy quality metrics and then consider combinations of them.

Weighted Structural Complexity (WSC) is a generalization of policy size [9]. For an RBAC policy π of the above form, we define weighted structural complexity by $\mathrm{WSC}(\pi) = w_1|R| + w_2|UA| + w_3|PA| + w_4|RH|$, where $|s|$ is the size (cardinality) of set s, and the w_i are user-specified weights. For an RBAC policy with direct assignment, the definition is the same except with an additional summand $w_5|DA|$.

Interpretability is a policy quality metric measures how well the roles in the policy can be characterized (interpreted) in terms of user attributes. Specifically, we quantify policy interpretability as *attribute mismatch*, which measures how well the sets of members of the roles can be characterized using expressions over user attributes. An *attribute expression* e is a function from the set A of attributes to sets of values. A user u *satisfies* an attribute expression e iff $(\forall a \in A.\ f(u, a) \in e(a))$. For example, if $A = \{dept, level\}$, the function e with $e(dept) = \{\mathrm{CS}\}$ and $e(level) = \{2, 3\}$ is an attribute expression, which can be written with syntactic sugar as $dept \in \{\mathrm{CS}\} \wedge level \in \{2, 3\}$. We refer to the set $e(a)$ as the conjunct for attribute a. Let $[\![e]\!]$ denote the set of users that satisfy e. For an attribute expression e and a set U' of users, the *mismatch* of e and U', denoted $\mathrm{mismatch}(e, U')$, is the size of the symmetric difference of $[\![e]\!]$ and U', where the symmetric difference of sets s_1 and s_2 is $s_1 \ominus s_2 = (s_1 \setminus s_2) \cup (s_2 \setminus s_1)$. The *attribute mismatch* of a role r, denoted $\mathrm{AM}(r)$, is $\min_{e \in E} \mathrm{mismatch}(e, \mathrm{assignedU}(r))$, where E is the set of all attribute expressions, and $\mathrm{assignedU}(r) = \{u \mid \langle u, r\rangle \in UA\}$. The *attribute mismatch* of an RBAC policy π (with or without direct assignment) is $\mathrm{AM}(\pi) = \sum_{r \in R} \mathrm{AM}(r)$. We define policy interpretability INT as attribute mismatch, i.e., $\mathrm{INT}(\pi) = \mathrm{AM}(\pi)$.

Compound policy quality metrics take multiple aspects of policy quality into account. One approach is to combine multiple policy quality metrics using a weighted sum; however, the choice of weights may be difficult or arbitrary. We combine metrics by Cartesian product, with lexicographic ordering on the tuples. Let $\text{INT-WSC}(\pi) = \langle \mathrm{INT}(\pi), \mathrm{WSC}(\pi)\rangle$ and $\text{WSC-INT}(\pi) = \langle \mathrm{WSC}(\pi), \mathrm{INT}(\pi)\rangle$.

Role Mining from ACLs.

The problem of *role mining from ACLs* is: given an ACL policy π_a and a policy quality metric Q, find an RBAC policy π_r that is consistent with π_a and has the best quality, according to Q, among policies consistent with π_a. The problem of *role mining with direct assignment from ACLs* is the same except that π_r is an RBAC policy with direct assignment.

Role Mining from ACLs and User Attributes.

The problem of *role mining from ACLs and user attributes* (with or without direct assignment) is the same as for role mining from ACLs, except that the input also includes user-attribute data, which may be used in the policy quality metric.

Our algorithms produce RBAC policies in which role membership is always defined by explicit user-role assignment, even when the current membership of a role can be characterized exactly by an attribute expression. In practice, assigning users to roles fully automatically based on user attributes might be risky; requiring explicit user-role assignments by an administrator is safer. The administrator's effort can be reduced by an algorithm that suggests appropriate roles for new users, based on their attributes. For example, we can compute and store a best-fit attribute expression e_r for each role r, i.e., an attribute expression that minimizes the attribute mismatch for r. When a new user u is added to the access control system, the system suggests that u be made a member of the roles for which u satisfies the best-fit attribute expression, and it presents these suggested roles for u in descending order of the attribute mismatch. This allows good suggestions even in the presence of noise.

3. ALGORITHMS

This section presents our role mining algorithms. In general, they compute only approximate solutions to the role-mining problem: the generated RBAC policy is always consistent with the given ACL policy, but it does not always have the best possible quality. This is a common limitation of role mining algorithms, because computing an optimal solution is NP-hard for policy quality metrics of interest [9].

3.1 Elimination Algorithm

Our *elimination algorithm* has three phases. Phase 1, role generation, generates a candidate role hierarchy that contains all "interesting" candidate roles. Phase 2, role elim-

```
    // Create initial roles.
1:  InitRole ← ∅
2:  permSets ← ⋃_{u∈U} {p ∈ P | ⟨u, p⟩ ∈ UP}
3:  for ps in permSets \ {∅}
4:     r = new Role()
5:     InitRole ← InitRole ∪ {r}
6:     PA ← PA ∪ ({r} × ps)
7:  end for

    // Compute all intersections of initial roles.
8:  R ← ∅
9:  for r in InitRole
10:    InitRole ← InitRole \ {r}
11:    for r′ in InitRole
12:       P ← assignedP(r) ∩ assignedP(r′)
13:       if ¬empty(P)∧ ∄r″ ∈ R. assignedP(r″) = P
14:          r″ = new Role()
15:          PA ← PA ∪ ({r″} × P)
16:          R ← R ∪ {r″}
17:       end if
18:    end for
19:    for r′ in R
20:       P ← assignedP(r) ∩ assignedP(r′)
21:       if ¬empty(P)∧ ∄r″ ∈ R. assignedP(r″) = P
22:          r″ = new Role()
23:          PA ← PA ∪ ({r″} × P)
24:          R ← R ∪ {r″}
25:       end if
26:    end for
27: end for
28: R ← R ∪ InitRole
```

Figure 1: Role generation, step 1: compute candidate roles.

ination, removes roles from the candidate role hierarchy if the removal preserves consistency with the given ACL policy and improves policy quality. Phase 3, role restoration, adds some removed roles back to the policy, if this improves policy quality.

Phase 1: Role Generation.

Our algorithm for role generation is based closely on CompleteMiner [14], although for increased scalability, we could easily substitute FastMiner [14] or the FP-Tree approach [5, 10]. Roles are characterized primarily by the set of permissions assigned to the role. An *initial role* has a set of permissions that contains all permissions assigned to some user. A *candidate role* has a set of permissions obtained by intersecting the permission sets of an arbitrary number of initial roles. As argued in [14], in the absence of other information on which to base the construction of candidate roles, this method generates all interesting candidate roles. Pseudo-code for this construction appears in Figure 1. It is essentially the same as the pseudo-code for CompleteMiner in [14]. It uses the functions $\text{assignedP}(r) = \{p \in P \mid \langle r, p\rangle \in PA\}$ and $\text{assignedU}(r) = \{u \in U \mid \langle u, r\rangle \in UA\}$.

CompleteMiner does not produce a role hierarchy. Our algorithm computes a role inheritance relation with the maximum amount of inheritance: a candidate role r_p inherits from another role r_c whenever the permissions of r_p are a superset of the permissions of r_c. Furthermore, when that

```
    // Initialize variables. Assign users to roles.
1:  UA ← ∅; RH ← ∅
2:  for u in U
3:     P ← {p ∈ P | ⟨u, p⟩ ∈ UP}
4:     for r in R
5:        if authP(r) ⊆ P
6:           UA ← UA ∪ {⟨u, r⟩}
7:        end if
8:     end for
9:  end for

    // Add inheritance edges, and eliminate inherited
    // permissions and members from UA and PA.
10: for r in R
11:    parents ← {r′ ∈ R | ⟨r, r′⟩ ∈ RH}     // parents of r
12:    for r′ in R \ {r}
13:       if authP(r′) ⊆ authP(r)
14:        ∧ ∀r″ ∈ parents. authP(r′) ⊈ authP(r″)
15:          RH ← RH ∪ {⟨r, r′⟩}
16:          for ⟨r, p⟩ in PA
17:             if p ∈ authP(r′)
18:                PA ← PA \ {⟨r, p⟩}
19:             end if
20:          end for
21:          for ⟨u, r′⟩ in UA
22:             if u ∈ assignedU(r)
23:                UA ← UA \ {⟨u, r′⟩}
24:             end if
25:          end for
26:          for r″ in parents
27:             if authP(r″) ⊈ authP(r′)
28:                RH ← RH \ {⟨r, r″⟩}
29:             end if
30:          end for
31:       end if
32:    end for
33: end for
```

Figure 2: Role generation, step 2: construct role hierarchy, based on R and PA from step 1.

inheritance relation is introduced, the permissions inherited by r_p from r_c are removed from the permissions explicitly assigned to r_p by PA, and the members inherited by r_c from r_p are removed from the members explicitly assigned to r_c by UA. Pseudo-code appears in Figure 2. It uses functions $\text{authP}(r) = \{p \in P \mid \exists r' \in R.\ \langle r, r'\rangle \in RH^* \wedge \langle r', p\rangle \in PA\}$ and $\text{authU}(r) = \{u \in U \mid \exists r' \in R.\ \langle r', r\rangle \in RH^* \wedge \langle u, r'\rangle \in UA\}$, where RH^* is the reflective transitive closure of RH.

A role hierarchy has *full inheritance* if every two roles that can be related by the inheritance relation are related by it, i.e., $\forall r, r' \in R.\ \text{authP}(r) \supseteq \text{authP}(r') \wedge \text{authU}(r) \subseteq \text{authU}(r') \implies \langle r, r'\rangle \in RH^*$. Guo *et al.* call this property *completeness* [4].

All of our algorithms generate RBAC policies with full inheritance. Although relaxing this requirement would allow our algorithms to achieve better policy quality in some cases, we impose this requirement, because in the absence of other information, all of these possible inheritance relationships are equally plausible, so removing any of them risks removing some that are semantically meaningful and desirable.

Phase 2: Role Elimination.

Roughly, the role elimination phase removes roles from the candidate role hierarchy if the removal preserves consistency with the given ACL policy and improves policy quality. When a role r is removed, the role hierarchy is adjusted to preserve inheritance relations between parents and children of r, and the user assignment and permission assignment are adjusted to explicitly assign to other roles the members and permissions that they previously inherited from r.

The order in which roles are considered for removal is important, because it may lead to different RBAC policies in the end. We control this ordering with a *role quality metric* Q_{role}, which maps roles to an ordered set, with the interpretation that large values denote high quality (note: this is opposite to the interpretation of the ordering for policy quality metrics). Low-quality roles are considered for removal first. The algorithm is parameterized by the choice of role quality metric. We consider three basic role quality metrics and then consider combinations of them.

Clustered size measures how well user permissions are clustered in the role. A first attempt at formulating such a metric might simply be the total number of *UP* pairs (i.e., elements of the *UP* relation) that are covered by the role, or, equivalently but with the metric normalized to be in the range $[0, 1]$, the fraction of all *UP* pairs covered by the role. However, such a metric would give the same rating to a role r_1 that covers one permission for each of 10 users and a role r_2 that covers 5 permissions for each of 2 users, even though r_2 is preferable; for example, if all of the users have exactly 5 permissions, then the two users in r_2 would not need to belong to any other roles, while all of the users in r_1 would need to belong to other roles as well. To take this into account, we define the clustered size metric to equal the fraction of the permissions of the role's members that are covered by this role; formally,

$$\text{assignedUP}(r) = \{\langle u, p\rangle \in UP \mid u \in \text{assignedU}(r) \wedge p \in \text{assignedP}(r)\}$$
$$\text{clsSz}(r) = |\text{assignedUP}(r)| \div |\{\langle u, p\rangle \in UP \mid u \in \text{assignedU}(r)\}|$$

The numerator considers assigned users and permissions, instead of authorized users and permissions, so that a role gets credit only for the *UP* pairs that it covers by itself, not for *UP* pairs covered by its ancestors or descendants.

Attribute fitness measures how well the set of members of a role can be characterized (interpreted) in terms of user attributes. It is based on attribute mismatch, defined in Section 2, normalized to be in the range $[0, 1]$ and subtracted from 1 so that higher values of the metric indicate higher quality; formally, $\text{attrFit}(r) = 1 - \frac{\text{AM}(r)}{|\text{assignedU}(r)|}$.

Redundancy measures how many other roles also cover the *UP* pairs covered by a role. Removing a role with higher redundancy is less likely to prevent subsequent removal of other roles, so we eliminate roles with higher redundancy first. Values of the redundancy metric are pairs, with lexicographic order. The redundancy of role r is the negative of the minimum, over *UP* pairs $\langle u, p\rangle$ covered by r, of the number of other removable roles that cover $\langle u, p\rangle$ (we take the negative so that roles with more redundancy have lower quality and hence get considered for removed first).

$$\text{authUP}(r) = \{\langle u, p\rangle \in UP \mid u \in \text{authU}(r) \wedge p \in \text{authP}(r)\}$$
$$\text{redun}(\langle u, p\rangle) = |\{r \in R \mid \langle u, p\rangle \in \text{authUP}(r) \wedge \text{removable}(r)\}|$$
$$\text{redun}(r) = -\min_{\langle u,p\rangle \in \text{authUP}(r)}(\text{redun}(\langle u, p\rangle))$$

```
1:  π ← policy produced by role generation
2:  q ← Q_pol(π)
3:  workList ← list containing removable roles in π
4:  changed ← true
5:  while ¬empty(workList) ∧ changed
6:      sort workList in ascending order by Q_role
7:      changed ← false
8:      for r in workList
9:          if ¬removable(r)
10:             remove r from workList
11:         else
12:             π′ ← removeRole(π, r)
13:             q′ ← Q_pol(π′)
14:             if q′ < δq
15:                 π ← π′
16:                 q ← q′
17:                 changed ← true
18:                 remove r from workList
19:             end if
20:         end if
21:     end for
22: end while

function removeRole(π, r)
23: ⟨U, P, R, UA, PA, RH⟩ ← π
24: R ← R \ {r}
25: for ⟨r1, r⟩ in RH
26:     RH ← RH \ {⟨r1, r⟩}
27:     for ⟨r, r2⟩ in RH
28:         if ⟨r1, r2⟩ ∉ RH*
29:             RH ← RH ∪ {⟨r1, r2⟩}
30:         end if
31:     end for
32:     for ⟨r, p⟩ in PA
33:         if p ∉ authP(r1)
34:             PA ← PA ∪ {⟨r1, p⟩}
35:         end if
36:     end for
37: end for
38: for ⟨r, r2⟩ in RH
39:     RH ← RH \ {⟨r, r2⟩}
40:     for ⟨r, u⟩ in UA
41:         if u ∉ authU(r2)
42:             UA ← UA ∪ {⟨r2, u⟩}
43:         end if
44:     end for
45: end for
46: return ⟨U, P, R, UA, PA, RH⟩
```

Figure 3: Role elimination.

Compound role quality metrics can be formed in the same ways as compound policy quality metrics, e.g., max(clsSz, attrFit).

Our algorithm may remove a role even if the removal worsens policy quality slightly. Specifically, we introduce a *quality change tolerance* δ, with $\delta \geq 1$, and we remove a role if the quality Q' of the RBAC policy resulting from the removal is related to the quality Q of the current RBAC policy by $Q' < \delta Q$ (recall that, for policy quality metrics, smaller values are better). Choosing $\delta > 1$ partially compensates for the fact that a purely greedy approach to policy quality improvement is not an optimal strategy.

Pseudo-code for role elimination appears in Figure 3. It is parameterized by a policy quality metric Q_{pol}, a role quality metric Q_{role}, and a quality change tolerance δ. A role is *removable* if every *UP*-pair covered by r is covered by at least one other role currently in the policy; formally,

$$\text{removable}(r) = \forall \langle u, p\rangle \in \text{authUP}(r).\ \exists r' \in R.\ r' \neq r \wedge \langle u, p\rangle \in \text{authUP}(r')$$

A removable role can be removed while preserving consistency with the given ACL policy. The removeRole function removes a role r, adjusts the role hierarchy to preserve inheritance relations between parents and children of r, and adjusts the user assignment and permission assignment to explicitly assign to other roles the members and permissions that they previously inherited from r. The removability test in line 9 is necessary because a role that is initially removable might become unremovable, due to other removals. The quality of each role is computed only in line 6, immediately before sorting the worklist. Role quality metrics may change as roles are removed and hence are re-computed each time line 6 is executed.

Phase 3: Role Restoration.

Phase 3 restores removed roles when this improves policy quality. Specifically, it considers each removed role r, in the same order that the roles were removed, and restores r if this improves the policy quality. Pseudo-code to restore a role appears in Figure 4. It uses the relation $\prec$ defined by $r \prec r' = \text{authP}(r) \subset \text{authP}(r')$. It makes r a child of roles r' such that $r \prec r' \wedge \neg\exists r'' \in R.\ r \prec r'' \prec r'$, makes r a parent of roles r' such that $r' \prec r \wedge \neg\exists r'' \in R.\ r' \prec r'' \prec r$, and adjusts the permission assignment, user assignment, and inheritance relations of roles related to r to eliminate redundancy.

Direct User-Permission Assignment.

If direct user-permission assignment is allowed, we add a final phase that replaces roles with direct assignment if that improves policy quality. Pseudo-code appears in Figure 5; variable π initially contains the policy produced by phase 3, which contains no direct assignments, i.e., $DA = \emptyset$.

Determining Algorithm Parameters.

Different choices of role quality metric Q_{role} and quality change tolerance δ may give the best results for different datasets, so we enclose the algorithm in a loop that tries all combinations of the following values for those parameters and returns the result from the best combination: Q_{role} in $\{\langle \text{redun}, \text{clsSz}\rangle, \langle \max(\text{attrFit}, \text{clsSz}), \text{redun}\rangle\}$, and δ in $\{1, 1.001, 1.002\}$. We also experimented with sum(clsSz, attrFit) for Q_{role}, and with larger values for δ, but that did not improve the results.

3.2 Selection Algorithm

Our *selection algorithm* works in the opposite way as the elimination based algorithm. Specifically, it starts with an empty policy and repeatedly adds candidate roles to the policy. The selection algorithm is parameterized by a role quality metric. In phase 1, candidate roles are generated as in the elimination algorithm (see Figure 1). In phase 2, candidate roles are added to the RBAC policy in order of descending role quality, until the RBAC policy is consistent with the given ACL policy. Phase 3 performs pruning: for each role r in the policy in the reverse order that the roles were added, checks whether the role is removable, and if so, whether removing it improves policy quality, and if so, removes it.

```
function restoreRole(π, r)
1:  ⟨U, P, R, UA, PA, RH⟩ ← π
2:  for r′ in R
3:      if r ≺ r′ ∧ ¬∃r″ ∈ R. r ≺ r″ ≺ r′
            // make r a child of r′
4:          assignedP(r′) ← assignedP(r′) \ authP(r)
5:          assignedU(r) ← assignedU(r) \ authU(r′)
6:          RH ← RH ∪ {⟨r′, r⟩}
7:          for r″ in R such that ⟨r′, r″⟩ ∈ RH   // children of r′
8:              if r″ ≺ r
                    // remove r″ as a child of r′. r″ will be
                    // a child of r and a grandchild of r′
9:                  RH ← RH \ {⟨r′, r″⟩}
10:             end if
11:         end for
12:     end if
13:     if r′ ≺ r ∧ ¬∃r″ ∈ R. r′ ≺ r″ ≺ r
            // make r a parent of r′
14:         assignedP(r) ← assignedP(r) \ authP(r′)
15:         assignedU(r′) ← assignedU(r′) \ authU(r)
16:         RH ← RH ∪ {⟨r, r′⟩}
17:         for r″ in R such that ⟨r″, r′⟩ ∈ RH // parents of r′
18:             if r ≺ r″
                    // remove r″ as a parent of r′. r″ will be
                    // a parent of r and a grandparent of r′
19:                 RH ← RH \ {⟨r″, r′⟩}
20:             end if
21:         end for
22:     end if
23: end for
24: R ← R ∪ {r}
25: return ⟨U, P, R, UA, PA, RH⟩
```

Figure 4: Restore role r to policy π.

```
1:  for r in R
2:      π1 ← removeRole(r)
3:      π2 ← π1 with all UP pairs in the given ACL policy
               that are not covered in π1 added to DA
4:      if Q_pol(π2) < δ Q_pol(π)
5:          π ← π2
6:      end if
7:  end for
```

Figure 5: Create direct user-permission assignment.

3.3 Complete Algorithm

Our *complete algorithm* has two phases. Phase 1 generates a hierarchical RBAC policy in exactly the same way as the elimination algorithm. Phase 2 is role removal. While the elimination algorithm heuristically takes a greedy approach to removals, the complete algorithm considers all subsets of the set of removable roles, to find the set of removals that produces the policy with the highest quality.

To avoid explicitly storing the set of sets of removable roles that have been explored so far, our role removal algorithm is expressed as a recursive search. Removal of one role may prevent subsequent removal of another role, but removals commute in the sense that, if it is possible to remove r_1 and then remove r_2, then it is also possible to remove r_2 and

Dataset	$\|U\|$	$\|P\|$	$\|UP\|$	high-fit N_a	high-fit AF	low-fit N_a	low-fit AF
healthcare	46	46	1486	20	1	5	0.79
domino	79	231	730	20	1	12	0.48
emea	35	3046	7220	20	1	6	0.56
apj	2044	1146	6841	40	0.94	10	0.57
firewall-1	365	709	31951	40	0.997	15	0.58
firewall-2	325	590	36428	40	1	10	0.50
americas-small	3477	1587	105205	50	0.95	9	0.36

Figure 6: Information about datasets. N_a is the number of attributes. AF is the attribute fit.

then remove r_1, and these two sequences of removals lead to the same policy. To ensure that the algorithm does not unnecessarily explore the same removals in multiple orders, we impose an arbitrary ordering on the removable roles, by storing them in a list R_{rmv}, and the algorithm considers only sequences of removals consistent with that ordering; in other words, it considers sequences of removals that correspond to subsequences (not necessarily contiguous) of R_{rmv}. The algorithm is parameterized by a policy quality metric Q_{pol}. The algorithm is complete in the following sense: if Q_{pol} is WSC, then the complete algorithm computes a policy that minimizes WSC among policies consistent with the given ACL policy; for other policy quality metrics Q_{pol}, the complete algorithm computes a policy that minimizes Q_{pol} among policies that are consistent with the given ACL policy and have full inheritance.

4. DATASETS

We know of no publicly available real ACL policies with user attribute data, so we use publicly available real ACL policies, described next, together with synthetic user attribute data, generated as described below.

The ACL policies are listed in Figure 6. They originate from Hewlett-Packard (HP) Labs [3]. The healthcare dataset was obtained by HP Labs from the U.S. Veteran's Administration, which has developed a comprehensive list of the healthcare permissions that may be assigned to licensed or certified providers. The domino data is from a set of user and access profiles for a Lotus Domino server. americas-small is a network access control policy from Cisco firewalls used to manage external business partner's access to HP's network. apj and emea are similar but smaller datasets. HP Labs produced the firewall-1 and firewall-2 datasets based on analysis of network connectivity permitted by Checkpoint firewall rules.

Generation of User Attribute Data.

Molloy *et al.* provide summary information about non-public user attribute data and ACL policies from three customers [11]; we exploit this to make our synthetic attribute data have some approximately realistic characteristics. Based on the information in the paper, we construct the following distributions: (a) for each customer i, we fit a Weibull distribution $card_i$ to the distribution of cardinalities of user attributes for that customer. (b) for each attribute of each customer, we fit a Zipf distribution to the distribution of values of that attribute (based on the information in [11, Figures 3-5]), to obtain a Zipf-distribution exponent for each attribute, and then we fit an exponential distribution *zipfExp* to the resulting distribution of Zipf-distribution exponents. The individual Zipf-exponents obtained from our measurements of the charts in [11, Figures 3-5] have considerable uncertainty, due to the limited information in those charts, but these uncertainties might average out to some extent, making the parameters of the exponential distribution *zipfExp* somewhat more robust.

Our algorithm for generating user attribute data is parameterized by an ACL policy and the desired number N_a of attributes. The algorithm has two phases. Phase 1 generates user attribute data for each attribute separately, independent of the ACLs. Phase 2 modifies the user attribute data to improve its fit with the ACLs. In more detail, phase 1 starts by identifying the customer i in [11] for which the number of users is closest to the number $|U|$ of users in the given ACL policy, and then, for each of the desired attributes, select a cardinality c_a from $card_i$ and a Zipf-exponent s_a from *zipfExp*. Next, the value of attribute a for each user is selected from a Zipf distribution with c_a elements and exponent s_a. We take all attribute values to be natural numbers interpreted as ranks in the Zipf distribution (0 is the most common value, 1 is the second most common value, etc.).

Phase 2 tries to reduce the attribute mismatch for each permission. Let U_p denote the set of users with permission p, i.e., $U_p = \{u \in U \mid \langle u, p\rangle \in UP\}$. For each permission p, we first compute an attribute expression e_p representing the least superset of U_p expressible as an attribute expression; e_p is given by $e_p(a) = \{f(u,a) \mid u \in U_p\}$. e_p may be a very loose upper bound on U_p, so we convert it to a lower bound on U_p by repeatedly removing an attribute value from a conjunct of e_p until $[\![e_p]\!] \subseteq U_p$; in each iteration, we remove the attribute value with the largest value of the metric m, where, for a value v in the conjunct for attribute a

$$m(v,a) = \begin{array}{l} |\{u \in U \mid f(u,a) = v \wedge u \notin U_p\}| \\ -|\{u \in U \mid f(u,a) = v \wedge u \in U(p)\}| \end{array}$$

Finally, we try to make the lower bound tighter as follows: for each user u in $U_p \setminus [\![e_p]\!]$, for each attribute a such that $f(u,a) \notin e_p(a)$, if adding $f(u,a)$ to $e_p(a)$ preserves the fact that $[\![e_p]\!] \subseteq U_p$, then add $f(u,a)$ to $e_p(a)$, otherwise try to modify f so that $f(u,a) \in e_p(a)$, by swapping the values of $f(u,a)$ and $f(u',a)$ for some other user u', provided the swap does not affect whether u' satisfies the attribute expressions already constructed for other permissions. Note that swapping values of attributes between users preserves the distribution of values of each attribute.

The *attribute fit* of the resulting attribute assignment is defined as $1 - \frac{1}{|UP|}\sum_{p \in P} \text{mismatch}([\![e_p]\!], U_p)$. For each dataset, we start with $N_a = 10$, generate user attribute data, and compute the attribute fit. If it is above 0.9, we stop, otherwise we increment the number of attributes by 10 and try again, until the attribute fit is above 0.9. We call the resulting user attribute data the *high-fit* user attribute data.

In practice, the available user attribute data will often have a lower attribute fit than 0.6, e.g., because some relevant user attributes are unavailable. Therefore, we also produce a version of the user attribute data with fewer attributes; specifically, we discard attributes one at a time, until the attribute fit drops below 0.6 (except we use a higher threshold of 0.8 for healthcare, otherwise N_a is very low). We call this the *low-fit* user attribute data.

Figure 6 contains information about the generated user

attribute data. Generation of user attribute data takes only a few minutes for small datasets, and it takes less than an hour for the largest dataset.

5. EXPERIMENTAL RESULTS

This section compares our algorithms with each other, compares the elimination algorithm (which is best among our algorithms) with prior work, and explores the effects of different policy quality metrics and role quality metrics.

Comparison of Elimination Algorithm with Hierarchical Miner and Graph Optimisation.

Figure 7 shows the WSC and interpretability (using the high-fit attribute data) of policies produced by the elimination algorithm and Hierarchical Miner (HM) [9] with policy quality metric WSC-INT and the WSC of policies produced Graph Optimisation (GO) [15] (modified slightly by Molloy *et al.* to use WSC as the policy quality metric). The weight vector for WSC contains all ones except that the weight for direct assignment is infinity (in other words, direct assignment is prohibited). In the comparison of eight role mining algorithms in [10] and the comparison of four role mining algorithms in [9], for this weight vector, the best WSC for every dataset is achieved by either HM or GO. *Figure 7 shows that the elimination algorithm achieves smaller or equal WSC than HM and GO on every dataset, while simultaneously achieving good policy interpretability* (Figure 11 shows that the elimination algorithm simultaneously achieves good results for both components of the policy quality metric). The WSC from HM and GO are 2.7% worse and 14.0% worse, respectively, averaged over the datasets, compared to the WSC from the elimination algorithm. The INT from HM is 46.3% worse, averaged over the datasets, compared to the INT from the elimination algorithm; this is not surprising, because HM does not consider user attributes or policy interpretability. The results for HM are computed from policies produced by HM that Molloy sent to us. The results for GO are from [9, Table VI] for all datasets except americas-small, which is not used in [9]; the results for GO for americas-small are from [10, Table 4].

On a PC with an Intel Core 2 Quad 2.66 GHz CPU (the processor has 4 cores, but our code is purely sequential), the elimination algorithm terminates in 30 seconds or less for all datasets except americas-small, which takes about 3.5 minutes. Running times for HM and GO are not reported in [15, 10, 9], and the implementations of HM and GO described in those papers are not publicly available. We fit curves to a graph of running time *vs.* $|UP|$ for the datasets in Figure 6 and found that a quadratic function fits well.

Figure 8 shows the result of our elimination algorithm when allowing direct assignments, with a WSC weight vector containing all ones. The results for HM are computed from policies producd by HM that Molloy sent us. The results for GO are from [9, Table VII] for all datasets except americas-small, which is not used in [9]; the results for GO for americas-small are from [10, Table 4]. The original GO does not consider direct assignment, but Molloy *et al.* extended GO to support it. *Figure 8 shows that the elimination algorithm achieves smaller WSC than HM and GO on every dataset, while simultaneously achieving good policy interpretability.* The WSC from HM and GO are 1.5% worse and 18.8% worse, respectively, averaged over the datasets, compared to the WSC from the elimination algorithm. The INT from HM is 15.2% worse, averaged over the datasets, compared to the INT from the elimination algorithm.

	Elimination		HM		GO
Dataset	INT	WSC	INT	WSC	WSC
healthcare	14	144	16	149	168
domino	21	404	30	418	413
emea	32	3709	92	3795	3888
apj	392	4248	411	4282	4600
firewall-1	48	1385	59	1426	1543
firewall-2	7	945	7	945	960
americas-small	214	6330	324	6710	9721

Figure 7: Comparison of elimination algorithm with policy quality metric WSC-INT, Hierarchical Miner, and Graph Optimisation, when direct user-permission assignment is prohibited.

	Elimination		HM		GO
Dataset	INT	WSC	INT	WSC	WSC
healthcare	9	140	10	142	168
domino	7	371	9	379	413
emea	36	3644	39	3693	3888
apj	130	3827	164	3862	4600
firewall-1	17	1340	21	1349	1543
firewall-2	4	944	4	944	960
americas-small	182	6214	198	6468	9721

Figure 8: Comparison of elimination algorithm with policy quality metric WSC-INT, Hierarchical Miner, and Graph Optimisation, when direct user-permission assignment is permitted.

Comparison of Elimination Algorithm with Attribute Miner.

Among prior work on role mining that takes policy interpretability into account, the most closely related is Molloy *et al.*'s work on Attribute Miner [9]. Figure 9 compares the elimination algorithm (using the redundancy role quality metric and $\delta = 1.001$) with Attribute Miner [9]. Molloy *et al.*'s implementation of Attribute Miner is not publicly available, so the results for Attribute Miner are from our own implementation of it. Attribute Miner is designed to optimize the policy quality metric Weighted Structural Complexity with Attributes (WSCA) [9]. WSCA differs from WSC in how the size of the user-role assignment is measured. In WSC, it is simply $|UA|$ or equivalently $\sum_{r \in R} |U(r)|$, where $U(r)$ is the membership (assigned users) of role r. In WSCA, if $U(r)$ can be characterized exactly by an attribute expression $D(r)$, the size of $D(r)$ (i.e., the number of conjuncts) is used instead of $|U(r)|$; otherwise, the geometric mean of $|U(r)|$ and $|[\![B(r)]\!]|$ is used instead of $|U(r)|$, where $B(r)$ is the attribute expression that is the least upper bound for $U(r)$. We have some reservations about WSCA: (1) use of the geometric mean of $|U(r)|$ and $|[\![B(r)]\!]|$ seems unintuitive, since it does not directly measure either the size or the interpretability of the role; (2) WSCA is very sensitive to whether a role can be characterized exactly by an attribute expression—a small change to the input data can significantly change the WSCA associated with a role, because $|D(r)|$ is often much smaller than $|U(r)|$; (3) as discussed at the end of Section 2, it might be safer to use attribute expressions to suggest role membership than to define role

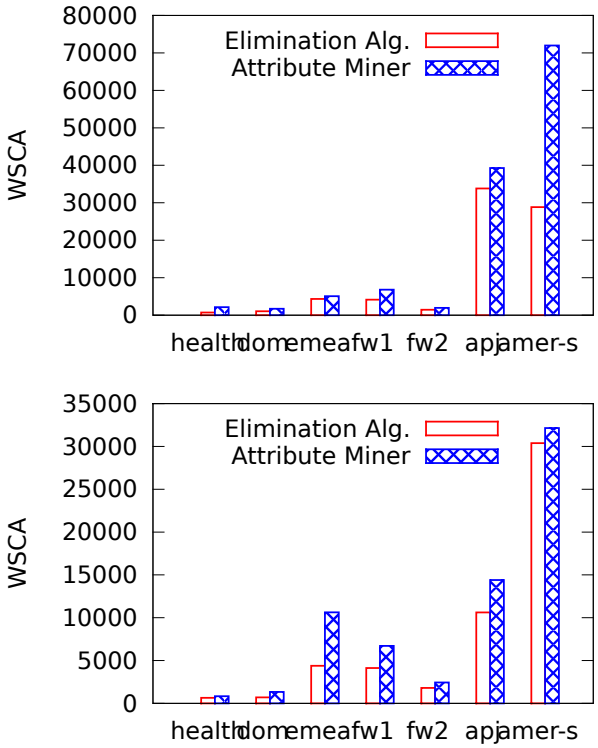

Figure 9: Comparison of elimination algorithm and Attribute Miner (AM). Names of datasets are abbreviated, e.g., fw1 abbreviates "firewall-1". The upper and lower graphs use the high-fit and low-fit user attribute data, respectively.

membership. Nevertheless, we use WSCA for this comparison, because Attribute Miner is designed to optimize WSCA and would probably fare poorly in a comparison based on INT-WSC.

Attribute Miner, as described in [9] uses attribute expressions that are conjunctions of positive literals over Boolean attributes. We implemented a generalized version of Attribute Miner that uses attribute expressions of the form described in Section 2. This involves straightforward changes to the code that computes least upper bounds and to the definition of the size of an attribute expression, which is used in the definition of WSCA [9, Definition 13] and in the definition of the cost of an attribute role [9, Table III]. We define the size of an attribute expression e to be $\sum_{a \in A} |e(a)|$. Attribute Miner takes user attribute data and a set of candidate roles as input; we generate the set of candidate roles using Phase 1 of the elimination algorithm.

Figure 9 shows that the elimination algorithm achieves better WSCA than Attribute Miner on every dataset. With the high-fit attribute data, Attribute Miner is 78% worse, averaged over the datasets, i.e., the average of the ratios of the WSCA values obtained using the two algorithms is 1.78; the median of the ratios is 1.38. With the low-fit attribute data Attribute Miner is 57% worse, averaged over the datasets, i.e., the average of the ratios of the WSCA values obtained using the two algorithms is 1.57; the median of the ratios is 1.36.

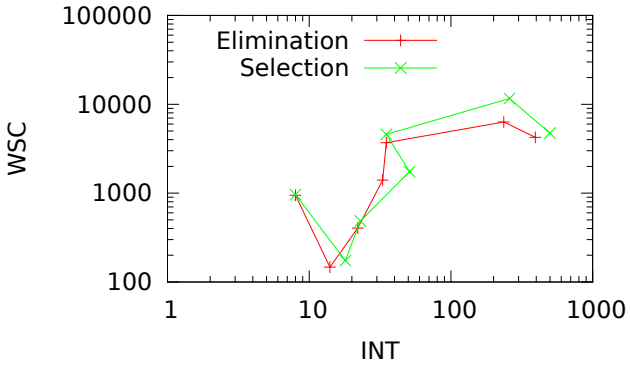

Figure 10: Results for elimination algorithm and selection algorithm, with policy quality metric INT-WSC. The clusters of points correspond, from left to right in the order they are connected, to the datasets in the following order: firewall-2 healthcare, domino, firewall-1, emea, americas-small, apj.

Comparison of Our Algorithms.

Figure 10 contains results for the elimination algorithm with the redundancy role quality metric and the selection algorithm with role quality metric max(attrFit, clsSz). We use INT-WSC as the policy quality metric for both algorithms. The weight vector for WSC contains all ones except that the weight for direct assignment is infinity (in other words, direct assignment is prohibited). Figure 10 shows that the elimination algorithm achieves the same or better results than the selection algorithm on both components of the policy quality metric for every dataset. We ran the complete algorithm on the smallest dataset, healthcare, with Q_{pol}=WSC. The result has WSC = 141, which is better than elimination algorithm (WSC = 144) and HM (WSC = 149). We started to run the complete algorithm on the second smallest dataset, domino, but we aborted it after 30 hours.

Effect of Policy Quality Metric in Elimination Algorithm.

Figure 11 compares the quality of policies produced by the elimination algorithm with policy quality metrics WSC-INT and INT-WSC, using the high-fit user attribute data. Recall that the elimination algorithm tries multiple role quality metrics Q_{role} and quality change tolerances δ; the tables also show the best combination of those parameters for each policy quality metric and each dataset. Surprisingly, for all of these datasets, it makes little or no difference whether priority is given to WSC or interpretability.

Effect of Role Quality Metric and Quality Change Tolerance in Elimination Algorithm.

We compared the results of the elimination algorithm with policy quality metric INT-WSC and four role quality metrics: redundancy, max(attrFit, clsSz), and the "reverse" of each of these, obtained by taking the negative of the value. The reverse orders exemplify a bad choice of role quality metric. We used $\delta = 1.0$ and policy quality metric WSC-INT with all four role quality metrics. Averaged over the datasets, using reverse-max(attrFit, clsSz) instead of max(attrFit, clsSz) worsens policy interpretability by 5.0% and WSC by 0.9%, and using reverse-redundancy instead of redundancy wors-

	WSC-INT				INT-WSC			
Dataset	INT	WSC	Q_{role}	δ	INT	WSC	Q_{role}	δ
healthcare	14	144	rdn	1.001	14	144	rdn	1.001
domino	21	404	max	1.001	21	404	max	1.001
emea	32	3709	max	1.000	32	3709	max	1.000
apj	392	4248	rdn	1.000	384	4331	rdn	1.002
firewall-1	48	1385	max	1.000	44	1419	max	1.003
firewall-2	7	945	max	1.000	7	945	max	1.000
amer-small	214	6330	max	1.000	180	6912	red	1.003

Figure 11: Comparison of two different policy quality metrics in elimination algorithm. "rdn" and "max" denote $\langle \text{redun}, \text{clsSz} \rangle$ **and** $\langle \max(\text{attrFit}, \text{clsSz}), \text{redun} \rangle$**, respectively.**

ens policy interpretability by 3.9% and WSC by 1.0%. This shows that the order in which roles are considered for removal has a small but non-negligible effect.

We also compared the results of the elimination algorithm using all six combinations of the two role quality metrics and three quality change tolerances specified in Section 3. We found that the combination Q_{role} = redun and $\delta = 1.001$ gives the best result or close to it—within 2% for WSC and interpretability—for every dataset in our experiments.

6. RELATED WORK

The literature on role mining is sizable, so we discuss only the most closely related work.

Vaidya *et al.*'s RoleMiner algorithm has two phases [14]. Phase 1 produces a set of candidate roles, each represented by a set of permissions. They give two algorithms for this: CompleteMiner, which we adopt as the first step in Phase 1 of our elimination algorithm, and FastMiner, which is similar to CompleteMiner but more scalable, because it considers only pairwise intersections of initial roles. Phase 2 prioritizes the candidate roles produced by Phase 1. The prioritized list of roles is the final result of the algorithm. The algorithm does not attempt to determine which candidate roles to include in such an RBAC policy, to produce a role inheritance relation, or to assign users to roles. In contrast, our algorithm addresses these issues in order to produce an RBAC policy. Vaidya *et al.* also developed algorithms for computing an RBAC policy with minimal $|R|$ that is consistent with a given ACL policy [13]. Lu *et al.* [7] present role mining algorithms that minimize either $|R|$ or $|UA| + |PA|$. None of these papers considers more general policy size metrics (such as WSC), role hierarchy, or interpretability of roles with respect to user attribute data.

Zhang *et al.*'s Graph Optimisation (GO) algorithm starts with each user's permission set as a candidate role, and repeatedly splits or merges roles when the transformation improves policy quality [15]. They do not consider interpretability of roles with respect to user attribute data. The data in Figures 7 and 8 show that the elimination algorithm achieves better WSC than GO does. The main reasons are: (1) GO performs role generation and role selection in a single phase, considering new candidate roles lazily according to a greedy heuristic, instead of eagerly generating all candidate roles in an initial phase; as a result, GO is faster, but it might fail to consider some useful roles; (2) it appears from the paper that GO does not explicitly control the order in which roles are considered for splitting and merging; and (3) GO never tries to eliminate roles.

Ene *et al.*'s role mining algorithms aim to minimize either $|R|$ or $|UA| + |PA|$ [3]. They do not consider policy interpretability with respect to user attribute data. Molloy *et al.* generalized the algorithm that aims to minimize $|UA|+|PA|$ so that it aims to minimize WSC instead, and they found that the modified algorithm performs well when the weight vector corresponds to the algorithm's original metric (i.e., when WSC equals $|UA| + |PA|$) but performs worse than GO and HM with other weight vectors [9], including the weight vectors used in our experiments.

Li *et al.*'s Dynamic Miner [6, 10] has three phases. Phase 1 generates a set of candidate roles. Phase 2 selects candidate roles to include in the RBAC policy, adding them to the policy in descending order of the estimated decrease in WSC achieved by adding the role (it is an estimate because the user-role assignment and role hierarchy are not known yet). Phase 3 constructs the user-role assignment and role hierarchy. Our selection algorithm is similar to Dynamic Miner, but more general, because it is parameterized by the role quality metric that controls the order in which roles are considered for selection, and, more importantly, it allows the role quality metric to take the role hierarchy and user-role assignment into account, because they are computed during the role selection phase. Molloy *et al.* found that Dynamic Miner generally produces worse WSC than HM and GO [10]. This is consistent with our finding that the selection algorithm generally produces worse results than the elimination algorithm.

Molloy *et al.*'s Hierarchical Miner (HM) has two phases. Phase 1 uses formal concept analysis to create a candidate role hierarchy consistent with a given ACL policy; phase 1 of the elimination algorithm is equivalent to phase 1 of HM. Phase 2 eliminates roles, removes their inheritance edges, or replaces them with direct user-permission assignment when this preserves consistency with the given ACL policy and lowers the WSC. The elimination algorithm achieves slightly better results than HM in our experiments. We believe this is mainly because the elimination algorithm uses a role quality metric to control the order in which roles are considered; the order in which roles are considered in HM is not explicitly controlled and depends on implementation details of a hashset library [8]. The use of a quality change tolerance and a role restoration phase also help the elimination algorithm achieve better results. Although phase 1 of HM produces a candidate role hierarchy with full inheritance, phase 2 of HM does not preserve this property; we plan to experiment with allowing similar deviations from full inheritance in the elimination algorithm, which should allow better results for policy quality. HM does not consider policy interpretability with respect to user attribute data.

Molloy *et al.*'s Attribute Miner (AM) has two phases. Phase 1 produces a set of candidate normal roles and a set of candidate attribute roles (i.e., roles whose membership is defined by an attribute expression). Phase 2 greedily selects normal roles and attribute roles for inclusion in the policy in descending order of the role's benefit-to-cost ratio, which is an estimate of the role's effect on the policy's WSCA. The elimination algorithm is more flexible than AM, since it can easily be used with any policy quality metric, and it achieves significantly better results than AM even for AM's target policy quality metric, namely, WSCA. We believe the

main reason for this is that the elimination approach (i.e., repeatedly remove roles) generally yields better results than the selection approach (i.e., repeatedly add roles), as we saw in the comparison of the elimination algorithm with our selection algorithm in Section 5, and as noted above in the discussion of Dynamic Miner.

Colantonio *et al.* propose two metrics to measure the interpretability of roles [1]. Their approach relies on an *activity tree*, describing the hierarchical structure of business activities (business processes), and an *organization unit tree*, describing the hierarchical structure of the organization. It also assumes knowledge of which permissions are required for each activity and of the assignment of users to organizational units. The *activity-spread* of a role measures the dispersion within the activity tree of the activities enabled by the role's permissions. The *organization-unit-spread* of a role measures the "dispersion" within the organization unit tree of the role's members. Roles with low activity-spread and low organization-unit-spread are considered to be more meaningful. These metrics are intuitively appealing and could be combined with metrics based on user attributes in our algorithms when the required information is available.

Colantonio *et al.* propose an approach to taking user attributes into account during role mining [2]. They first partition the set of users based on the values of selected attributes, and then perform role mining separately for each set of users in the partition (using the corresponding slice of the *UP* relation). Note that the role mining in the second step does not explicitly consider user attributes. They propose metrics that are used to select a set of attributes that provides the most meaningful partition of the users. Their paper does not consider metrics to directly evaluate the interpretability of the resulting roles or RBAC policies.

7. CONCLUSIONS AND FUTURE WORK

We presented a role mining algorithm, the elimination algorithm, that can easily be used to optimize a variety of policy quality metrics. In our experimental evaluation, using realistic datasets, it achieves equal or better results than previously proposed algorithms.

One direction for future work is to consider other metrics for policy interpretability, e.g., metrics that consider heterogeneity of users in different roles as well homogeneity of users in the same role [12]. Another direction is to improve scalability. We are exploring use of a scalable clustering algorithm or graph partitioning algorithm to decompose an ACL policy into subpolicies that can be role-mined separately; the metric that guides clustering or partitioning is designed to minimize policy quality loss due to the decomposition.

Acknowledgment.

We thank Ian Molloy for helpful comments and for sending us policies produced by Hierarchical Miner.

8. 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 *SACMAT '09: Proc. 14th ACM symposium on Access control models and technologies*, pages 85–94. ACM, 2009.

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

[3] A. Ene, W. G. Horne, N. Milosavljevic, P. Rao, R. Schreiber, and R. E. Tarjan. Fast exact and heuristic methods for role minimization problems. In *Proc. 13th ACM Symposium on Access Control Models and Technologies (SACMAT 2008)*, pages 1–10, 2008.

[4] 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 '08)*, pages 237–246, 2008.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000)*, pages 1–12. ACM, 2000.

[6] N. Li, T. Li, I. Molloy, Q. Wang, E. Bertino, S. Calo, and J. Lobo. Role mining for engineering and optimizing role based access control systems. Technical Report 2007-60, CERIAS, Purdue University, November 2007.

[7] H. Lu, J. Vaidya, and V. Atluri. Optimal boolean matrix decomposition: Application to role engineering. In *Proc. 24th International Conference on Data Engineering (ICDE)*, pages 297–306, 2008.

[8] I. Molloy. Private communication, Dec. 2011.

[9] 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):36, 2010.

[10] I. Molloy, N. Li, T. Li, Z. Mao, Q. Wang, and J. Lobo. Evaluating role mining algorithms. In *Proc. 14th ACM Symposium on Access Control Models and Technologies (SACMAT 2009)*, pages 95–104, 2009.

[11] I. Molloy, J. Lobo, and S. Chari. Adversaries' holy grail: Access control analytics. In *Proc. First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS 2011)*, pages 52–59, 2011.

[12] G. J. Szekely and M. L. Rizzo. Hierarchical clustering via joint between-within distances: Extending ward's minimum variance method. *J. Classification*, 22(2):151–183, 2005.

[13] J. Vaidya, V. Atluri, and Q. Guo. The role mining problem: Finding a minimal descriptive set of roles. In *Proc. 12th ACM Symposium on Access Control Models and Technologies (SACMAT 2007)*, pages 175–184, 2007.

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

[15] D. Zhang, K. Ramamohanarao, and T. Ebringer. Role engineering using graph optimisation. In *Proc. 12th ACM Symposium on Access Control Models and Technologies (SACMAT 2007)*, pages 139–144, 2007.