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 representa-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

 ACM 12, June 20–22, 2012, Newark, New Jersey, USA.
Copyright 2012 ACM 978-1-4503-1295-0/12/06 ...$10.00.

...ions, 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;
• 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 \( (U, P, UP) \), 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 \( (U, P, R, UA, PA, RH) \), 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, \( (r, r') \in RH \) 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 \( (U, P, R, UA, PA, RH, DA) \), 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 \( (A, f) \), 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 \( \pi \) of the above form, we define weighted structural complexity by \( WSC(\pi) = w_1|R| + w_2|UA| + w_3|PA| + w_4|RH| \), 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 \( A = \{ \text{dept}, \text{level} \} \), the function \( e \) with \( e(\text{dept}) = \{ \text{CS} \} \) and \( e(\text{level}) = \{ 2, 3 \} \) is an attribute expression, which can be written with syntactic sugar as \( \text{dept} \in \{ \text{CS} \} \land \text{level} \in \{ 2, 3 \} \).

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 \ {∅} do
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 do
10: InitRole ← InitRole \ \{ r \}
11: for r’ in InitRole do
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 do
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.

Initation, 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 Com-
pleteMiner [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 per-
missions 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 in-
formation 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
especially the same as the pseudo-code for CompleteMiner
in [14]. It uses the functions assignedP(r) = \{ p ∈ P | ⟨r, p⟩ ∈ PA \} and assignedU(r) = \{ u ∈ U | ⟨u, r⟩ ∈ UA \}.

CompleteMiner does not produce a role hierarchy. Our al-
gorithm computes a role inheritance relation with the max-
imum 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 do
3: P ← \{ p ∈ P | ⟨u, p⟩ ∈ UP \}
4: for r in R do
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 do
11: parents ← \{ r’ ∈ R | ⟨r, r’⟩ ∈ RH \} // parents of r
12: for r’ in R \ \{ r \} do
13: if authP(r’) ⊆ authP(r)
14: ∧ ∀ r’’ in parents. authP(r’’) ∉ authP(r’’)
15: RH ← RH ∪ \{ ⟨r, r’⟩ \}
16: for ⟨r, p⟩ in PA do
17: if p ∈ authP(r’)
18: PA ← PA \ \{ ⟨r, p⟩ \}
19: end if
20: end for
21: for ⟨u, r’⟩ in UA do
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
authP(r) = \{ p ∈ P | ∃ r’ ∈ R. ⟨r, r’⟩ ∈ RH* ∧ ⟨r’, p⟩ ∈ PA \} and authU(r) = \{ u ∈ U | ∃ r’ ∈ R. ⟨r’, r⟩ ∈ RH* ∧ ⟨u, r’⟩ ∈ UA \}, where RH* is the reflec-
tive 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., ∀ r, r’ ∈ R. authP(r) ≥ authP(r’) ∧ authU(r) ≤ authU(r’’) =⇒ ⟨r, r’⟩ ∈ 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 rela-
tionships are equally plausible, so removing any of them risks
removing some that are semantically meaningful and desir-
able.
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 \( Q_r \), 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,

\[
\text{assignedUP}(r) = \{(u,p) \in U P \mid u \in \text{assignedU}(r) \}
\]

\[
\text{clsSz}(r) = |\text{assignedUP}(r)| \div |\{(u,p) \in U P \mid u \in \text{assignedU}(r)\}|
\]

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, \( \text{attrFit}(r) = 1 - \frac{\Delta \text{Mi}(r)}{\text{assignU}(r)} \).

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 \( (u,p) \) covered by \( r \), of the number of other removable roles that cover \( (u,p) \) (we take the negative so that roles with more redundancy have lower quality and hence get considered for removal first).

\[
\text{authUP}(r) = \{(u,p) \in U P \mid u \in \text{authU}(r) \land p \in \text{authP}(r)\}
\]

\[
\text{redun}(u,p) = |\{r \in R \mid (u,p) \in \text{authUP}(r) \land \text{removable}(r)\}|
\]

\[
\text{redun}(r) = -\min_{(u,p) \in \text{authUP}(r)}(\text{redun}(u,p))
\]

\[
\pi \leftarrow \text{policy produced by role generation}
\]

\[
q \leftarrow Q_{pol}(\pi)
\]

\[
\text{workList} \leftarrow \text{list containing removable roles in } \pi
\]

\[
\text{changed} \leftarrow \text{true}
\]

\[
\text{while } \neg \text{empty(workList)} \land \text{changed}
\]

\[
\text{sort workList in ascending order by } Q_{role}
\]

\[
\text{changed} \leftarrow \text{false}
\]

\[
\text{for } r \in \text{workList}
\]

\[
\text{if } \neg \text{removable}(r)
\]

\[
\text{remove } r \text{ from workList}
\]

\[
\text{else}
\]

\[
\pi' \leftarrow \text{removeRole}(\pi, r)
\]

\[
q' \leftarrow Q_{pol}(\pi')
\]

\[
\text{if } q' < \delta q
\]

\[
\pi \leftarrow \pi'
\]

\[
q \leftarrow q'
\]

\[
\text{changed} \leftarrow \text{true}
\]

\[
\text{remove } r \text{ from workList}
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

\[
\text{end while}
\]

\[
\text{function removeRole}(\pi, r)
\]

\[
(U, P, R, UA, PA, RH) \leftarrow \pi
\]

\[
R \leftarrow R \setminus \{r\}
\]

\[
\text{for } (r_1, r) \text{ in } RH
\]

\[
RH \leftarrow RH \setminus \{(r_1, r)\}
\]

\[
\text{for } (r, r_2) \text{ in } RH
\]

\[
\text{if } (r_1, r_2) \notin RH^*
\]

\[
RH \leftarrow RH \cup \{(r_1, r_2)\}
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

\[
\text{empty(workList)} \lor \text{changed}
\]

\[
\text{sort workList in ascending order by } Q_{role}
\]

\[
\text{changed} \leftarrow \text{false}
\]

\[
\text{for } r \in \text{workList}
\]

\[
\text{if } \neg \text{removable}(r)
\]

\[
\text{remove } r \text{ from workList}
\]

\[
\text{else}
\]

\[
\pi' \leftarrow \text{removeRole}(\pi, r)
\]

\[
q' \leftarrow Q_{pol}(\pi')
\]

\[
\text{if } q' < \delta q
\]

\[
\pi \leftarrow \pi'
\]

\[
q \leftarrow q'
\]

\[
\text{changed} \leftarrow \text{true}
\]

\[
\text{remove } r \text{ from workList}
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

\[
\text{return } (U, P, R, UA, PA, RH)
\]

\[
\text{Figure 3: Role elimination.}
\]

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

Our algorithm may remove a role even if the removal worsens policy quality slightly. Specifically, we introduce a quality change tolerance \( \delta \), with \( \delta > 1 \), 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 $\delta$. A role is removable if every UP-pair covered by $r$ is covered by at least one other role currently in the policy; formally,

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

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 $r \prec r' = \text{authP}(r) \subseteq \text{authP}(r')$. It makes $r$ a child of roles $r'$ such that $r \prec r' \land \neg \exists r'' \in R. r \prec r'' \prec r'$, makes $r$ a parent of roles $r'$ such that $r' \prec r \land \neg \exists r'' \in R. r' \prec r'' \prec r$, 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 $\pi$ 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 $\delta$ 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 $\{(\text{redun, clsSz}), (\text{max atrFit, clsSz}), \text{redun}\}$, and $\delta$ in $\{1, 1.001, 1.002\}$. We also experimented with $\text{sum}(\text{clsSz, atrFit})$ for $Q_{role}$, and with larger values for $\delta$, 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.

**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 then remove $r_1$. In the algorithm, we order the roles so that if a role $r'$ is removable without satisfying other removals, any role $r''$ that it removes will be removable without satisfying other removals. The variable $DA$ is the set of removable roles, to find the set of removals that produces the policy with the highest quality.

**Figure 4: Restore role $r$ to policy $\pi$.**

1: for $r$ in $R$
2: \hspace{1em} $\pi_1 \leftarrow \text{removeRole}(r)$
3: \hspace{1em} $\pi_2 \leftarrow \pi_1$ with all UP pairs in the given ACL policy that are not covered in $\pi_1$ added to $DA$
4: \hspace{1em} if $Q_{pol}(\pi_2) < \delta Q_{pol}(\pi)$
5: \hspace{2em} $\pi \leftarrow \pi_2$
6: \hspace{1em} end if
7: end for

**Figure 5: Create direct user-permission assignment.**

phase 3 produces pruning: for each role $r$ in the policy in the reverse order that the roles are added, checks whether the role is removable, and if so, whether removing it improves policy quality, and if so, removes it.
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 $R_{rev}$, 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 $R_{rev}$. 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 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 firewalls.

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., $U_p = \{ u \in U \mid (u,p) \in UP \}$. 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 $a$

$$m(v, a) = \frac{v \cap U_a}{U_a} - \frac{v \cap U_{\neg a}}{U_{\neg a}},$$

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 $f(u,a) \notin [e_p]$, add $f(u,a)$ to $e_p(a)$ 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 $1 - \frac{1}{|UP|} \sum_{p \in P} mismatch([e_p], U_p)$. 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.
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. |U| 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 produced by HM that Molloy sent to 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.

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 δ = 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 WSCA, it is simply |U|A or equivalently ∑r∈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
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 \( \sum_{a \in A} |c(a)| \). 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.

**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 \( \max(\text{attrFit}, \text{clsSz}) \). 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 \( Q_{pol} = \text{WSC} \). 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 \( \delta \); 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(\text{attrFit}, \text{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(\text{attrFit}, \text{clsSz}) instead of max(\text{attrFit}, \text{clsSz}) worsens policy interpretability by 5.0% and WSC by 0.9%, and using reverse-redundancy instead of redundancy wors-
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 $Q_{role} = \text{redun}$ and $\delta = 1.001$ gives the best result or close to it—within 2% for WSC and interpretability—for every dataset in our experiments.

### Dataset WSC-INT INT-WSC

<table>
<thead>
<tr>
<th>Dataset</th>
<th>INT</th>
<th>WSC</th>
<th>$Q_{role}$</th>
<th>$\delta$</th>
<th>INT</th>
<th>WSC</th>
<th>$Q_{role}$</th>
<th>$\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>healthcare</td>
<td>14</td>
<td>144</td>
<td>rdn 1.001</td>
<td>1.001</td>
<td>14</td>
<td>144</td>
<td>rdn 1.001</td>
<td>1.001</td>
</tr>
<tr>
<td>domino</td>
<td>21</td>
<td>404</td>
<td>max 1.000</td>
<td>1.001</td>
<td>21</td>
<td>404</td>
<td>max 1.000</td>
<td>1.001</td>
</tr>
<tr>
<td>emea</td>
<td>32</td>
<td>3709</td>
<td>max 1.000</td>
<td>1.000</td>
<td>32</td>
<td>3709</td>
<td>max 1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>apj</td>
<td>392</td>
<td>4248</td>
<td>rdn 1.000</td>
<td>1.002</td>
<td>384</td>
<td>4331</td>
<td>rdn 1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>firewall-1</td>
<td>48</td>
<td>1385</td>
<td>max 1.000</td>
<td>1.003</td>
<td>44</td>
<td>1419</td>
<td>max 1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>firewall-2</td>
<td>7</td>
<td>945</td>
<td>max 1.000</td>
<td>1.000</td>
<td>7</td>
<td>945</td>
<td>max 1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>amer-small</td>
<td>214</td>
<td>6330</td>
<td>max 1.000</td>
<td>1.003</td>
<td>180</td>
<td>6912</td>
<td>red 1.003</td>
<td>1.003</td>
</tr>
</tbody>
</table>

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

### 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 $|UA| + |PA|$ [3]. 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 rule’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


